A DISSERTATION

ON

ANALYSING EFFECT OF REFACTORING ON SOFTWARE MAINTAINABILITY USING OBJECT ORIENTED METRICS

Submitted in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY
In
SOFTWARE TECHNOLOGY

Submitted by
Nirmala
University Roll No. 2K15/SWT/511

Under the Esteemed Guidance of
Dr. Ruchika Malhotra
Associate Head & Associate Professor,
Department of Computer Science & Engineering, DTU

2015-2018

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY,
DELHI– 110042, INDIA

DELHI TECHNOLOGICAL UNIVERSITY

DELHI-110042

DECLARATION

I hereby declare that the thesis entitled **"ANALYSING EFFECT OF REFACTORING ON SOFTWARE MAINTAINABILITY USING OBJECT ORIENTED METRICS"** which is being submitted to the **Delhi Technological University**, in partial fulfillment of the requirements for the award of the degree of **Master of Technology in Software Technology** is an authentic work carried out by me. The material contained in this thesis has not been submitted to any university or institution for the award of any degree.

DATE:

SIGNATURE:

Nirmala
2K15/SWT/511

DELHI TECHNOLOGICAL UNIVERSITY

DELHI-110042

CERTIFICATE

This is to certify that thesis entitled **“ANALYSING EFFECT OF REFACTORING ON SOFTWARE MAINTAINABILITY USING OBJECT ORIENTED METRICS”,** is a bona fide work done by Ms. Nirmala (Roll No: 2K15/SWT/511) in partial fulfillment of the requirements for the award of **Master of Technology Degree in Software Technology** at Delhi Technological University, Delhi, is an authentic work carried out by her under my supervision and guidance. The content embodied in this thesis has not been submitted by him earlier to any University or Institution for the award of any Degree or Diploma to the best of my knowledge and belief.

DATE:

SIGNATURE:

Dr. RUCHIKA MALHOTRA
ASSOCIATE HEAD & ASSOCIATE PROFESSOR,
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING.
DELHI TECHNOLOGICAL UNIVERSITY, DELHI 110042

ACKNOWLEDGEMENT

I am presenting my work on **"ANALYSING EFFECT OF REFACTORING ON SOFTWARE MAINTAINABILITY USING OBJECT ORIENTED METRICS"** with a lot of pleasure and satisfaction. I take this opportunity to express my deep sense of gratitude and respect towards my guide **Dr. Ruchika Malhotra**. I am very much indebted to her for her generosity, expertise and guidance I have received from her while working on this project. Without her support and timely guidance the completion of the project would have seemed a far-fetched dream. In this respect, I find myself lucky to have my guide. She has guided not only with the subject matter, but also taught the proper style and techniques of documentation and presentation. Besides my guides, I would like to thank entire teaching and non-teaching staff in the Department of Computer Science & Engineering, DTU for all their help during my tenure at DTU. Kudos to all my friends at DTU for thought provoking discussion and making stay very pleasant. I am also thankful to the SAMSUNG who has provided me opportunity to enroll in the M.Tech Program and to gain knowledge through this program. This curriculum provided me the knowledge and an opportunity to grow in various domains of computer science.

Nirmala
2K15/SWT/511

ABSTRACT

Software maintainability is the ease with which a software components can be modified to rectify the defects or their cause, repair or supplant broken or exhausted segments without replacing the working parts, prevent unexpected working condition, maximize a product's useful life, maximize efficiency, reliability, and safety, meet new requirements, make future maintenance easier, or cope with a changed environment.

For vast programming frameworks, the maintenance stage has longer term than all the past life-cycle stages taken together, causing significantly more exertion. The time spent and exertion required to keep software product operational after deployment is exceptionally critical and expends to 40-70% of the total cost of the whole life cycle. Nice measure of software maintainability can enable better dealing in the maintenance stage exertion. In past writing, analysts and experts have proposed few machine learning calculations with a target to anticipate programming viability and assess them.

Maintainability model as described by S. Counsell [6] is used as base in this study. Maintenance is very important phase of software life cycle. And many researchers [1,2,3,4,5,16] have already shared their findings about object oriented metric and code refactoring has direct impact on maintainability. As per past literature, Maintainability [6,7,11] , C&K metrics [4] , other multiple OO metrics and Code-refactoring have some relation with each other.

Since refactoring was first investigated as a maintenance discipline in the late 1990's, it has moved toward becoming a vital part of an engineer's tool-set and generated numerous refactoring experimental studies. Seventy-two types of refactoring were described by Martin Fowler, in his book [14], which includes renaming, conditional-statements, structural modifications and many more coding areas.

The objective of this study is to calculate object oriented metrics[1,3,4,5,6,7,9,10,11] which can be further used with JArchitct tool and ref-finder to correlate it with software maintainability . In order to study, software repository of android application CALENDAR [22] is used. The motive is to generate a data set of a repository to measure the maintainability index of software based on object-oriented software metrics using the JHawk [19] tool. And using JArchitect2018.1.0 (demo) tool [20] for code smells and refactoring extraction. Then analysing the relation between object oriented metrics change with maintainability index and impact of refactoring on maintainability. The result shows that the dataset is successfully generated, and code-refactoring is more in those components which have low maintainability index.

TABLE OF CONTENTS

DECLARATION ii
CERTIFICATE iii
ACKNOWLEGEMENT iv
ABSTRACT v
TABLE OF CONTENTS vii
LIST OF TABLES ix
LIST OF FIGURES x
LIST OF ACRONYMS..........xii

Chapter 1: Introduction 1
Chapter 2: Literature Review 6
Chapter 3: Research Background..........10
3.1 Data set generation..........10
3.1.1 JHAWK Tool..........12
3.1.2 OO Metrics into consideration..........14
CKJM Metrics 15
1. Weighted Methods Per Class (WMC) 16
2. Depth of Inheritance Tree (DIT)..........16
3. Number of Children (NOC)..........16
4. Coupling between Object Classes (CBO)..........16
5. Response for a Class (RFC)..........17
6. Lack of Cohesion of Methods (LCOM)..........17
3.2 Jhawk Metrics 17
3.3 Software Maintainability model..........18
Chapter 4: Research Methodlogy..........19
4.1 Preprocessing of Data 20
4.2 Jhawk Tool Outputs 20
4.3 Object Oriented metric capturing process..........33
4.4 Refactoring Dataset construction 33
4.4.1 Code-Refactoring..........34
- Rename..........34
- Move Class..........35
- Extract Method (Long Methods)..........35
- Extract Classes and SuperClasses 35
- Replace Conditions with Polymorphism (too many conditional statements) 36

4.4.2 JArchitect Tool....37
4.5 Results....40
4.5.1 DONUT-ECLAIR Correlation....41
4.5.2 ÉCLAIR-FROYO Correlation....41
4.5.3 FROYO-GINGERBREAD OS Correlation....42
4.5.4 GINGERBREAD-ICS OS Correlation....43
4.5.5 ICS-JB OS Correlation....44
4.5.6 JB-KITKAT OS Correlation....44
Chapter 5: Conclusion & Future Work....46
Bibliography....49

LIST OF TABLES

Table 3. 1 Repository links of all the OS versions of android app Calendar 10

Table 3. 2 Dataset for different OS versions of android application Calendar 11

Table 4. 1 Code-Smell and other Metrics captured 39

Table 5. 1 System Data 46

LIST OF FIGURES

Figure 3. 1 JHAWK Tool 14

Figure 3. 2 Jhawk Tool- OO Metrics generation 15

Figure 4. 1 Dataset construction and analysis-process 19

Figure 4. 2 Jhawk output for Donut 21

Figure 4. 3 Jhawk output for Eclair 22

Figure 4. 4 Jhawk output for Froyo 23

Figure 4. 5 Jhawk output for Gingerbread 24

Figure 4. 6 Jhawk output for ICS 25

Figure 4. 7 Jhawk output for JB 26

Figure 4. 8 Jhawk output for Kitkat 27

Figure 4. 9 Jhawk output for Lollipop 28

Figure 4. 10 Jhawk output for Marshmallow 29

Figure 4. 11 Jhawk output for Nougat 30

Figure 4. 12 Jhawk output for Oreo 30

Figure 4. 13 Jhawk output for Pie 31

Figure 4. 14 JArchitect Tool 37

Figure 4. 15 Metric-View of Calendar-Lollipop Version 38

Figure 4. 16 Code-Smells captured by JArchitect of Calendar-Lollipop Version 39

Figure 4. 17 MI and Refactoring of Donut & Éclair 41

Figure 4. 18 Eclair-Froyo : Refactoring vs MI 42

Figure 4. 19 Froyo-Gingerbread : Refactoring vs MI 43

Figure 4. 20 Gingerbread-ICS : Refactoring vs MI 43

Figure 4. 21 ICS-JB : Refactoring vs MI 44

Figure 4. 22 JB-Kitkat : Refactoring vs MI 45

Figure 5. 1 : MI ,TCC,NOM,LOC(java statement) metric plot of 12 OS versions of Calendar app. 47

Figure 5. 2 : Total Refactoring in 6 OS versions of Calendar app.................................... 47

Chapter 1: Introduction

Software maintainability practicality implies the simplicity with which a product framework or segment can be adjusted to rectify flaws, enhance performance or different credits or adjust to a changed situation. The adjustment in the product is required to meet the changing necessities of clients which may emerge because of numerous reasons, for example, change in the innovation, and presentation of new equipment or upgrade of the requirement and so on. Delivering Software product which does not should be changed isn't just infeasible yet additionally exceptionally uneconomical. This procedure of changing the product which has been conveyed is called software maintenance. The measure of asset, exertion and time spent on software maintenance is considerably more than what is being spent on its before-deployment programming. Along these lines, creating a software product that is anything but difficult to maintain may conceivably spare substantial expenses and endeavours.

One of the primary methodologies in controlling maintenance cost is to screen programming measurements (metrics) amid the development stage. It involves enthusiasm to quantify different properties of programming configuration as far as inheritance, coupling, and cohesion and so on and anticipate its maintenance pattern based on their quantitative values. The issue of foreseeing the maintainability of software is broadly recognized in the business and much has been composed on how this can be anticipated by utilizing different algorithms and procedures at the season of development using software metrics [1, 3, 4, 5, 6, 7].

Studies have been led and found the solid connection between Object Oriented metrics and software maintainability. They have additionally discovered that these metrics can be utilized as indicators of maintenance effort. Exact forecast of software maintainability can be valuable as a result of the accompanying reasons: (a). It helps venture directors in looking at the efficiency and expenses among various undertakings. (b). It furnishes directors with data for all the more viably arranging the utilization of significant assets. (c). It helps administrators in taking imperative choice in regards to staff portion. (d). It controls about support process proficiency. (e). It helps in monitoring future support exertion. (f). The edge estimations of different metrics which definitely influence maintainability of software product can be checked and monitored in order to accomplish minimum upkeep cost. (g). It empowers the engineers to recognize the determinants of programming quality with the goal that they can enhance plan and coding. (h). It encourages specialists to enhance the nature of programming frameworks and in this manner upgrade support costs.

To quantify the maintainability we first discover the "change “. It is characterized as "how much measure of normal endeavours are required to include, change or erase existing classes". Software maintenance is vital as it expends 70% of the season of any item's life. In spite of this reality it is ineffectively overseen on the grounds that we truly don't have great measures of software maintainability. The crucial factor is expressed by Counsell [6] in the year 2015, that the coupling, defects and size have impact on maintainability.

To gauge the different highlights of oops, for example, inheritance, cohesion, coupling, memory distribution and so forth unique measurements are deliberately chosen. We have contemplated different metrics accessible in writing and chose just those software metrics that have a solid association with programming maintenance and utilized them while developing our model for expectation of question arranged programming maintainability[1].These metrics are related to effort per module, total cyclomatic complexity per module, size per module and MI per module.

When a software's source code is effectively understandable, the software is more maintainable, prompting decreased expenses and enabling valuable advancement assets to be utilized somewhere else. In the meantime, if the code is all around organized, new prerequisites can be presented more productively and with less issues. These two improvement errands, maintainability and upgrade, frequently struggle since new highlights, particularly those that don't fit neatly inside the first outline, result in an expanded support exertion. The re-factoring procedure expects to lessen this contention, by helping non damaging changes to the structure of the source code, keeping in mind the end goal to improve code clearness and viability.

Refactoring enhances non-functional traits of the product. Favorable circumstances incorporate enhanced code intelligibility and decreased many-sided quality; these can enhance source-code viability and make a more expressive inside engineering or protest model to enhance extensibility. Normally, refactoring applies a progression of institutionalized fundamental miniaturized scale refactoring, every one of

which is (more often than not) a little change in a PC program's source code that either protects the conduct of the product, or if nothing else does not adjust its conformance to utilitarian necessities. There are multiple refactoring techniques present , some of them are Rename , Move Class , Extract Method(Long Methods) , Extract Classes and Super-Classes , Replace Conditions with Polymorphism(too many conditional statements),Fields Removed ,Methods removed ,Classes Removed , Methods Direct Calling , Method indirect Calling and Classes with poor cohesion are also considered in this study. All these refactoring types have self-explanatory names.

In this work, we propose a dataset that we assembled using the JArchitect2018.1.0(demo) tool [20] for code smells extraction and the JHawk (starter) tool for source code metric calculation of a open source git-repository of CALENDAR(https://android.googlesource.com/platform/packages/apps/Calendar) from Android subsequent releases Donut to Lollipop.

Google Calendar is a time administration and planning logbook benefit created by Google. It ended up accessible in beta version in April 13, 2006, on the web and as portable applications for the Android and iOS. Google Calendar enables clients to make and alter occasions in it. Reminders is supported, which can be set on the basis of user preferences like, time days or months. Occasion areas can likewise be included, and different clients can be welcome to occasions. Application recovers dates of births from Google contacts and shows birthday cards on a yearly premise, and Holidays, a nation particular schedule that shows dates of unique events. Time to time, Google has included

usefulness that makes utilization of machine learning.Code smells are treated as the code-refactoring. Further the change in OO metrics and maintainability of software components is analysed w.r.t to refactoring which is captured using code-smells and naming convention change metric-rules of JArchitect tool.

Chapter 2: Literature Review

Several studies were done in the past to relate OO software metrics and refactoring with maintainability metric. Some of the key areas are discussed below.

Software Engineering Institute (SEI), Carnegie Mellon University [17] proposed the metric to measure the cost of maintainability based on source code,MI.This is an important element that permits the software engineer to have the capacity to foresee the maintenance effort while writing the code. MI is ascertained using polynomial equation that can be just computed dependent on the code lines, comments and complexity of the code also described in chapter3.Further based on this study, Hybrid intelligent Model for Software Maintenance prediction [7], MI is studied again and provided a maintainability prediction model based on evolutionary neural network. The suggested model is proved to have very good accuracy but is not that transparent to users.

S.Counsell , X.Liu ,S.Eidh,R.Tonelli, M.Marchesi,G.Concas and A.Murgia [6] , explored the MI metric and scrutinized the OO metric. Five metrics were used including coupling, defect and size. Coupling between objects (CBO) metric of Chidamber and kemerer[2] ,Fan-in(FIN) of a class, number of java statements in a class(NOS) and defects that each class has encountered were captured for three software projects and analyzed w.r.t MI. And huge correlations were found in the study.

S.Counsell, X.Liu ,S.Swift, J.Buckley,M.English,S.Herold,S.Eldh,A.Ermedahl [9] , proposed a refactoring type IEV as one of the very important when we see defect statistic. It was shown via study that IEV refactoring had been applied to those classes which were expected to be more inclined towards defects. IEV(Introduce explaining variable) and RCF(Remove control flag) are 2 refactoring type which were seemed to be related to defect-inclined classes.

István Kádár, Péter Heged˝us, Rudolf Ferenc and Tibor Gyimóthy [10,11],encouraged the investigation of code refactoring by providing an excessive open dataset of source code metrics and applied refactoring through several releases of 7 open-source systems. Results show that lower maintainability indeed triggers more code refactoring in practice and these refactoring significantly decrease complexity, code lines, coupling and clone metrics.[10] "A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability" paper presented a manually validated dataset of applied refactoring and source code metrics and maintainability of 7 open-source systems. It is a subset of [11].And found that Refiner had around 27% overall average precision on the subject systems, thus new – manually validated – subset has substantial added value allowing researchers to perform more accurate empirical investigations. Study answered, whether refactoring were really triggered by poor maintainability of the code, or by other aspects. The results show that source code elements subject to refactoring had significantly lower maintainability values (approximated by source code metric aggregation) than elements not affected by refactoring between two releases.

Birgit Geppert, Audris Mockus, and Frank Rößler [18], studied a refactoring on a part of a large legacy business communication product where protocol logic in the registration domain was restructured. And pose a number of hypotheses about the strategies and effects of the refactoring effort on aspects of changeability and measure the outcomes. The results of this case study show a significant decrease in customer reported defects and in effort needed to make changes.

Ruchika Malhotra and Anuradha Chug [13], provided a study which states that, refactoring is very tedious and might introduce errors if not implemented with utmost care, it is still advisable to frequently refactor the code to increase maintainability. Results of this study are useful to project managers in identifying the opportunities of refactoring while maintaining a perfect balance between reengineering and over-engineering.

Francesca Arcelli Fontana and Stefano Spinelli [12], Explored the code smells and relevant refactoring and their impact on software quality. Feature Envy, Long method , shotgun surgery and large class are the types of code smells considered in study and refactoring are applied accordingly. The monitored metric were WMC,LCOM,RFC,DAC,NOM and TCC.TCC increased for after all the refactoring , WMC increased only for Extract method same pattern is observed for RFC.LCOM decreased on all four types of refactoring. Hence Code smells result into refactoring which further effects the Quality metric of software.

Chapter 3: Research Background

Here, we will see the data collection process, tools used in our experiment, OO metrics generation etc.

3.1 Data set generation

In this study, OO Metrics were obtained using open source mobile OS – Android. 12 Android OS versions of an application namely Calendar. OS versions from donut,éclair,froyo,gingerbread,ICS,JB,kitkat,lollipop,marshmallow,nougat,oreo and pie are considered for generating the data sets.

Table 3. 1 Repository links of all the OS versions of android app Calendar

S.No.	OS Version	Repository link
1	DONUT	https://android.googlesource.com/platform/packages/apps/Calendar/+/donut-release
2	ECLAIR	https://android.googlesource.com/platform/packages/apps/Calendar/+/eclair-release
3	FROYO	https://android.googlesource.com/platform/packages/apps/Calendar/+/froyo-release
4	GINGERBREAD	https://android.googlesource.com/platform/packages/apps/Calendar/+/gingerbread-release
5	ICS	https://android.googlesource.com/platform/packages/apps/Calendar/+/ics-mr1-release
6	JB	https://android.googlesource.com/platform/packages/apps/Calendar/+/jb-release
7	KITKAT	https://android.googlesource.com/platform/packages/apps/Calendar/+/kitkat-release
8	LOLLIPOP	https://android.googlesource.com/platform/packages/apps/Calendar/+/lollipop-release
9	MARSHMAL	https://android.googlesource.com/platform/packages/apps/Calendar/+/marshmal

	LOW	low-release
10	NOUGAT	https://android.googlesource.com/platform/packages/apps/Calendar/+/nougat-release
11	OREO	https://android.googlesource.com/platform/packages/apps/Calendar/+/oreo-release
12	PIE	https://android.googlesource.com/platform/packages/apps/Calendar/+/pie-release

Source code is fetched from Google GIT repository [13] *(https://android.googlesource.com/platform/packages/apps/Calendar/)* for above subsequent OS versions of an application. The source code contains java files. First Android code is downloaded. JHAWK tool [14] is used to generate the object oriented metrics data set. JHawk is a static code analysis tool - i.e. it takes the source code of software project and calculates metrics based on numerous aspects of the code - for example volume, complexity, relationships between class and packages and relationships within classes and packages. This tool takes input of source code file & generated the component wise OO metrics as mentioned in Table 3.1.2.

Characteristics of different android application package with respect to Android releases are mentioned in Table 3.1 and Table 3.2 .

Table 3. 2 Dataset for different OS versions of android application Calendar

OS Versions	Total Classes
DONUT	**97**
ECLAIR	**107**
FROYO	**111**

GINGERBREAD	**123**
ICS	**220**
JB	**255**
KITKAT	**324**
LOLLIPOP	**326**
MARSHMALLOW	**326**
NOUGAT	**326**
OREO	**326**
PIE	**326**

GIT is open source versioning control system used for source code management task for Google android code. GIT as a distributed revision control system is aimed for speed, integrity of data and support for non-linear, distributed workflows. Google GIT Repository: https://android.googlesource.com/platform/packages/apps/...

Table 3.1.2 contains android app data sets with total class, total number Of classes having changes w.r.t. multiple Android release versions Calendar Application. Metrics were generated using JHAWK Tools ,downloaded from http://www.virtualmachinery.com/jhawkprod.html .

3.1.1 JHAWK Tool

JHawk is a static code investigation tool - i.e. it takes the source code of your undertaking and ascertains metrics dependent on various parts of the code - for instance size,complexity, connections among java-class and packages.

J-Hawk is a Java based open source system which can be added in any java application for testing.The thought is the designer needs to characterize module and its errands, by errands we mean a method inside the Java code. j-Hawk executes the modules and creates a graphical report which can be dissected to discover choke-point of any Java-coded application. J-Hawk gives it's own scripting language called \"hawk\" which is similar to C,C++ and UNIX Shell scripting. Henceforth the client can actualize the experiment effortlessly with hawk scripting.

Jhawk Tool [15] is a JAVA based automated tool which collects and reports various oo metric of a given version of android Operating System (OS). Matrices generation from Jhawk Tool depends completely on files of Java Project parsed to it.

Various studies [23] in the past have used this tool to carry out the research work on Java repository. Jhawk tool determines the multiple oo metrices including No. of Methods,LCOM, AVCC, NOS, HBUG, HEFF, UWCS, INST, PACK, RFC, CBO, MI, CCML, NLOC, RVF, F-IN, DIT, MINC, S-R, R-R, COH, LMC, LCOM2, MAXCC, HVOL, HIER, NQU, FOUT, Superclass, SIX, EXT, NSUP, TCC, NSUB, MPC, NCO, INTR, CCOM, Mod etc. It efficiently collects the data from java code repository under our research. Finally, the corresponding values of different metric suites are obtained by the system for each class files in the source code of android OS.

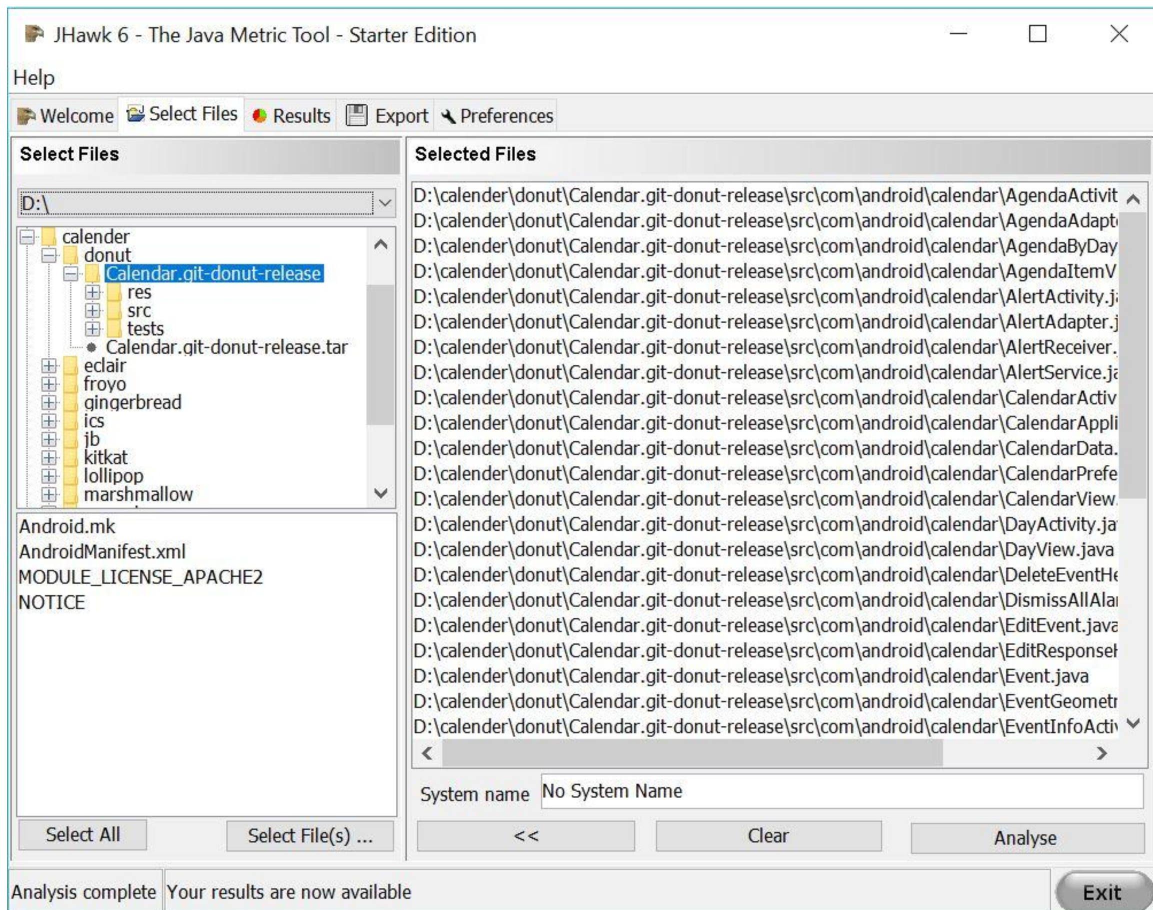


Figure 3. 1 JHAWK Tool

Install & configure GIT first, for source code of each version of the Android OS. Find the path of each android application on Google site: (*https://android.googlesource.com*) for corresponding TAGs i.e. Donut, Éclair, Froyo, Gingerbread,ICS,JB,Kitkat,Lollipop,Marshmallow,Nougat,Oreo and Pie. Now, download source code of each application for passing corresponding versions to Jhawk tool. Versioning can be seen through above GIT Tags. Figure 3.1.1 shown below is the tool home page after selecting the Donut code-folders.

3.1.2 OO Metrics into consideration

OO metrics is used to predict & evaluate the software's quality. OO metrics generated is used for MI & as an early indicator of externally visible attributes (like

cohesion, coupling, Encapsulation, inheritance etc.) CKJM metrics is the most popular used as OO Metrics.

OO Metrics were generated using Jhawk tool on each Java file. We provided the path of generated class files and downloaded source code to tool, and tool generated OO metrics for each of the classes of android application packages. Figure 3.1.2 display's the Jhawk tool captured OO metrics.

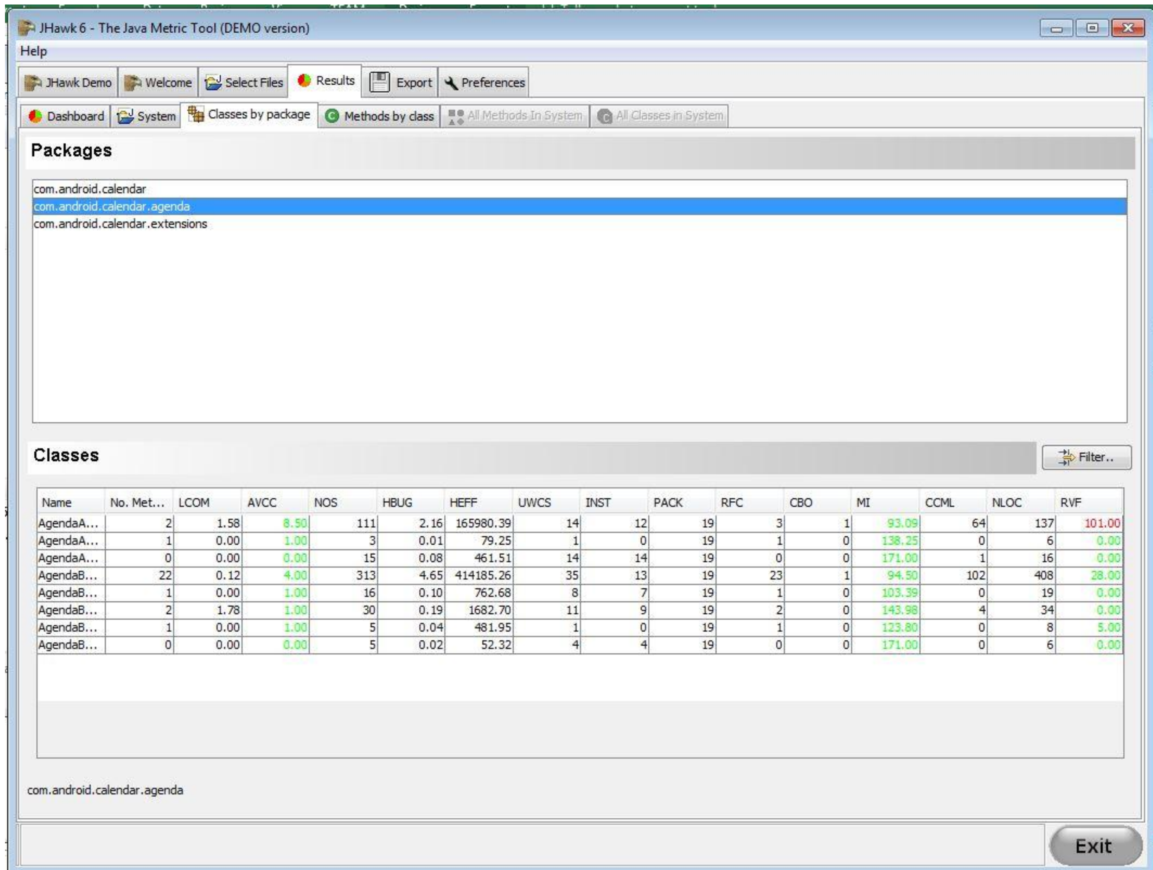


Figure 3. 2 Jhawk Tool- OO Metrics generation

CKJM Metrics

C&K [2] define the so called C&K metric suite. This metric suite offers informative insight whether developers are following OO principles in their design & development. This metrics helps managers to create higher style selections. C&K metrics

is incredibly standard among the researchers conjointly also and it’s the most well-known suite of measurements for OO software quality. C&K had projected six metrics. Following discussion describes its attributes:

1. Weighted Methods Per Class (WMC)

WMC is total number of the functions in a class. It measures the complexity of any class and it is can be checked by the cyclomatic complexity of the methods

2. Depth of Inheritance Tree (DIT)

DIT is the maximum inheritance level of the class from its base class.A low value of DIT is preferred as a high DIT indicates increase errors in the project.

3. Number of Children (NOC)

NOC is total number of immediate children of any class. It measures sub classes number that is inheriting the methods of its parent class. NOC shows absolute exertion required to test the class & its reuse.

A high NOC, a large no. of child class, indicates High reuse of a base-class. High NOC indicates lesser bugs in code.

4. Coupling between Object Classes (CBO)

CBO demonstrates coupling between the classes. If the object is utilizing any other object, at that point it is said to be coupled. A class is combined with another class if the techniques for one class are utilizing the strategies for inferior. An increase in CBO

demonstrates decline in class re-usability. Thus, the CBO for each class must be as less as would be prudent.

5. Response for a Class (RFC)

For any reaction to message, RFC is the count of function/method that are called. As RFC increases, testing effort also increases with testing arrangement develops. Plan multifaceted nature of a class increments with increment in RFC esteem and it ends up more earnestly to get it. On opposite side, its lower esteem speaks to more polymorphism. RFC values lies somewhere in the range of 0 and 50 for any class, it can increment up to 100 for certain cases relying upon undertaking.

6. Lack of Cohesion of Methods (LCOM)

LCOM metric speaks to level of equity between the methods. It demonstrates the level of cohesiveness in the software, for example way of structuring of the framework and measure of complex nature of the class. LCOM is subtraction of the quantity of methods combines whose resemblance is zero and tally count of method pairs whose similitude isn't zero. Along these lines, LCOM value ought to be kept Low and cohesion high.

3.2 Jhawk Metrics

Jhawk Tool captures more then 35 Object oriented metrics, it included Name of the Class, Weighted Methods Per Class, LCOM, Total Cyclomatic Complexity, Number of statements, Modifiers, Interfaces, Response for Class, Message passing, Coupling between objects, Maintainability Index (MI), Cohesion, (DIT)Depth of Inheritance Tree, Lines of code etc.

3.3 Software Maintainability model

In our study, the dependent variable is the MI and its Predicted value of classes & the OO metrics of the class is the independent variables. The objective of our study is to establish the relation of OO metrics, the change in OO metrics in subsequent version of class of operating system. We have used CKJM metrics with other OO metrics as independent variables. It is also calculated using JHAWK tool along with OO metrics generation. The metrics given by C&K [2] are summarized in Table 3.2. In figure 3.2.1, MI is the dependent variable which dependent on independent variables. Maintainability model used in Jhawk is as below:

MI = 171 - 3.42ln(aveE) - 0.23aveV(g') - 16.2ln(aveLOC) + (50 * sin(sqrt(2.46*aveCM)) [I]

Chapter 4: Research Methodlogy

We have conducted an empirical validation of correlation between MI and code refactoring of 12 releases of the android OS given in Table 3.1 1 using the following steps.

1. Pre-processing of android repository.
2. Generating OO metric change data set from the output of step 1.
3. Building JArchitect pprojects for each and every OS version under consideration.
4. Generating and mapping the Refactoring data of repository.
5. Plotting the refactoring data w.r.t to component MI of Calendar app and representing graphical results.

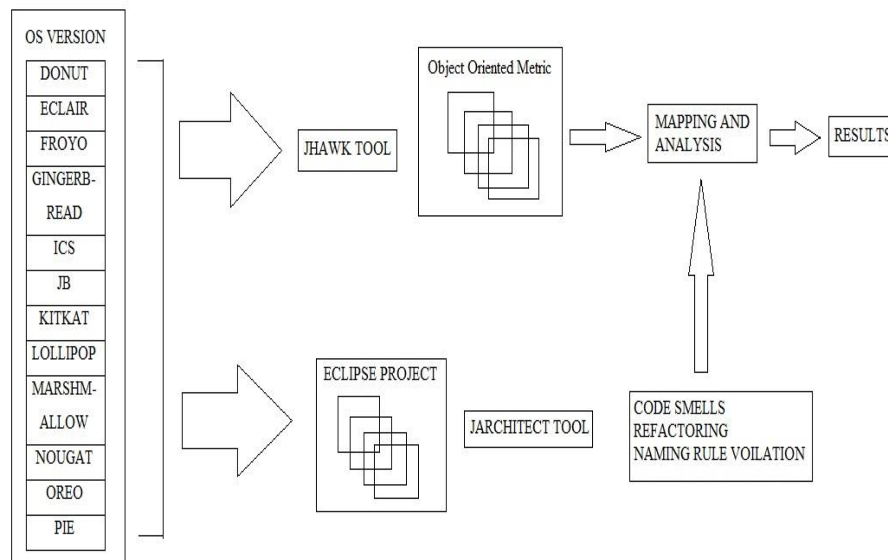


Figure 4. 1 Dataset construction and analysis-process

4.1 Preprocessing of Data

In this segment, we check the MI and refactoring metric generation techniques. MI model[I] is based on OO measurements utilizing Jhawk tool. The number of Lines of code(LCOM) in donut is 9588, eclair has 11093, froyo has 11697, gingerbread has 12549, ICS has 25410, JB has 28914, kitkat as 34208 and all later OS adaptation has 34281 classes from lollipop to pie. KitKat onwards, the OO metric information is relatively same, which is clear from Table 5.1. As the quantity of classes are relatively same, in working framework KitKat or more forms, we will discuss about Donut, Eclair, Froyo, Gingerbread, ICS, JB and KitKat OS versions of calendar application.

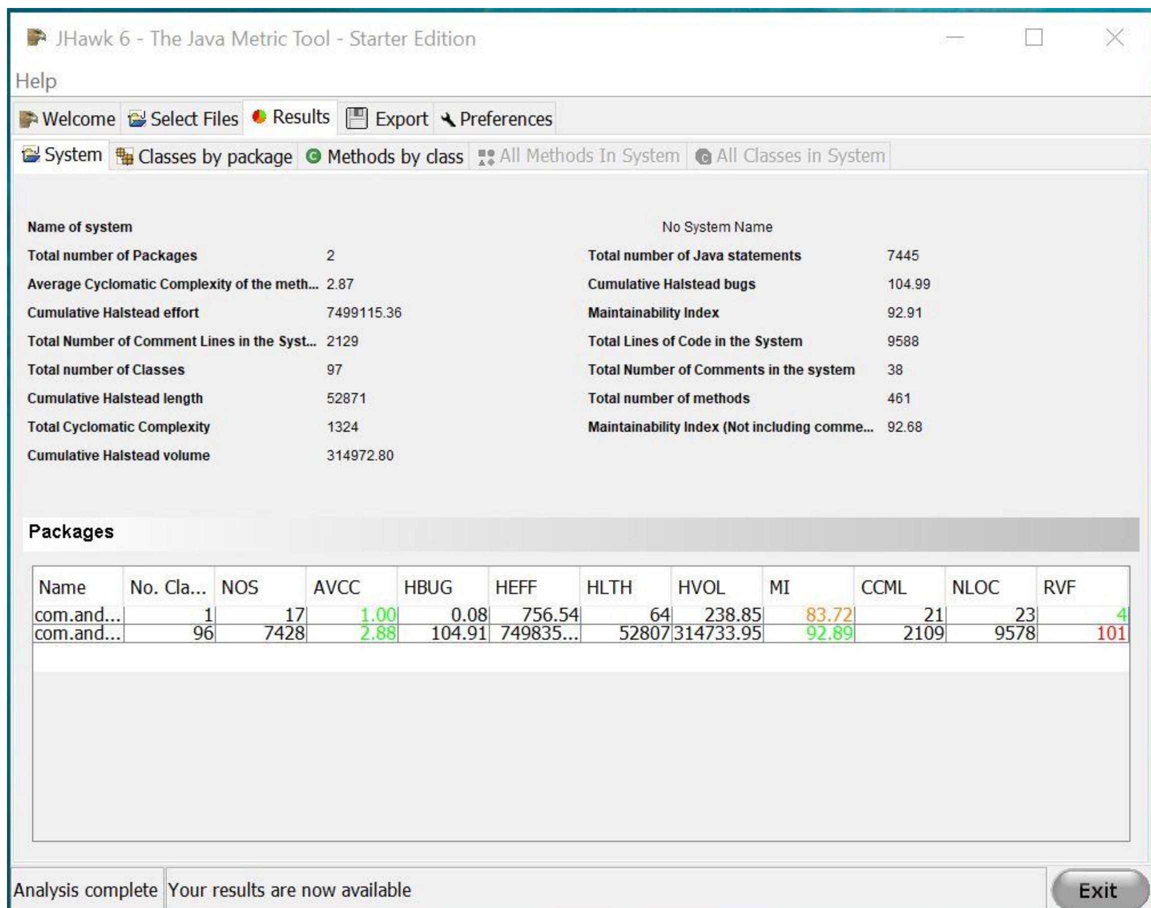
4.2 Jhawk Tool Outputs

In our study, MI metric of each class & the OO metrics of the class is focused. The objective of our study is to establish the relation of OO metrics, the MI metric and code refactoring in subsequent version of classes of calendar operating system. We have used CKJM metrics with other OO metrics as independent variables. It is also calculated using JHAWK tool along with OO metrics generation. The metrics given by C&K [2] are summarized in section 3.1 2.

1. Donut Operating System

As illustrated in Figure 4. 2, Donut version of Calendar was very small module , which is having only 97 java classes, therefore its overall maintainability is good when compared with other operating system versions. The total number of codes is 9588, and MI is 92.91. And as the size is less, the scope of refactoring also reduces. Table 5.1 1 shows that the refactoring is also lowest when Éclair version came.

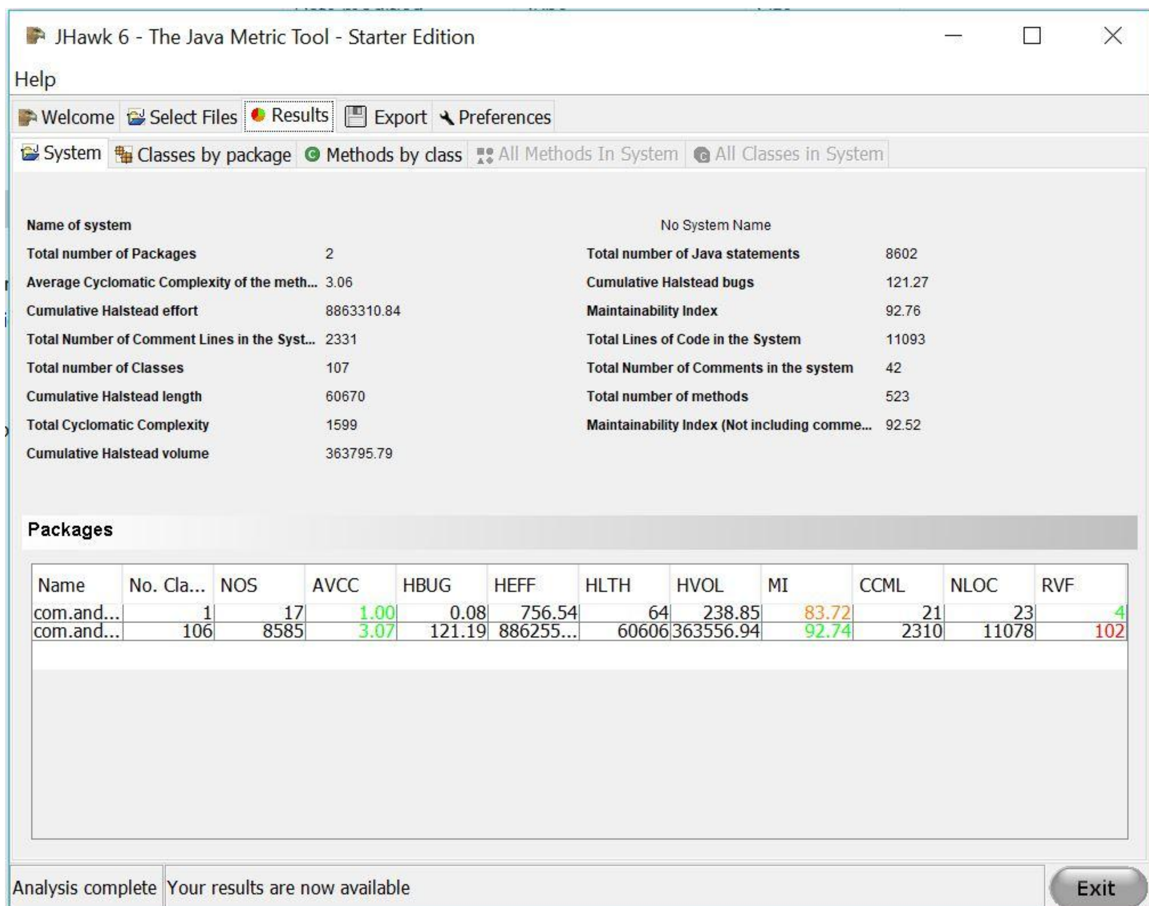
Figure 4. 2 Jhawk output for Donut



2. Eclair Operating System

Figure 4. 3 , is having Éclair system analysis of Calendar application. In MI metric value shown an improvement of 0.14 units, and as per data set, refactoring is also there. So our data-set is also inclined with Jhawk output.

Figure 4. 3 Jhawk output for Eclair

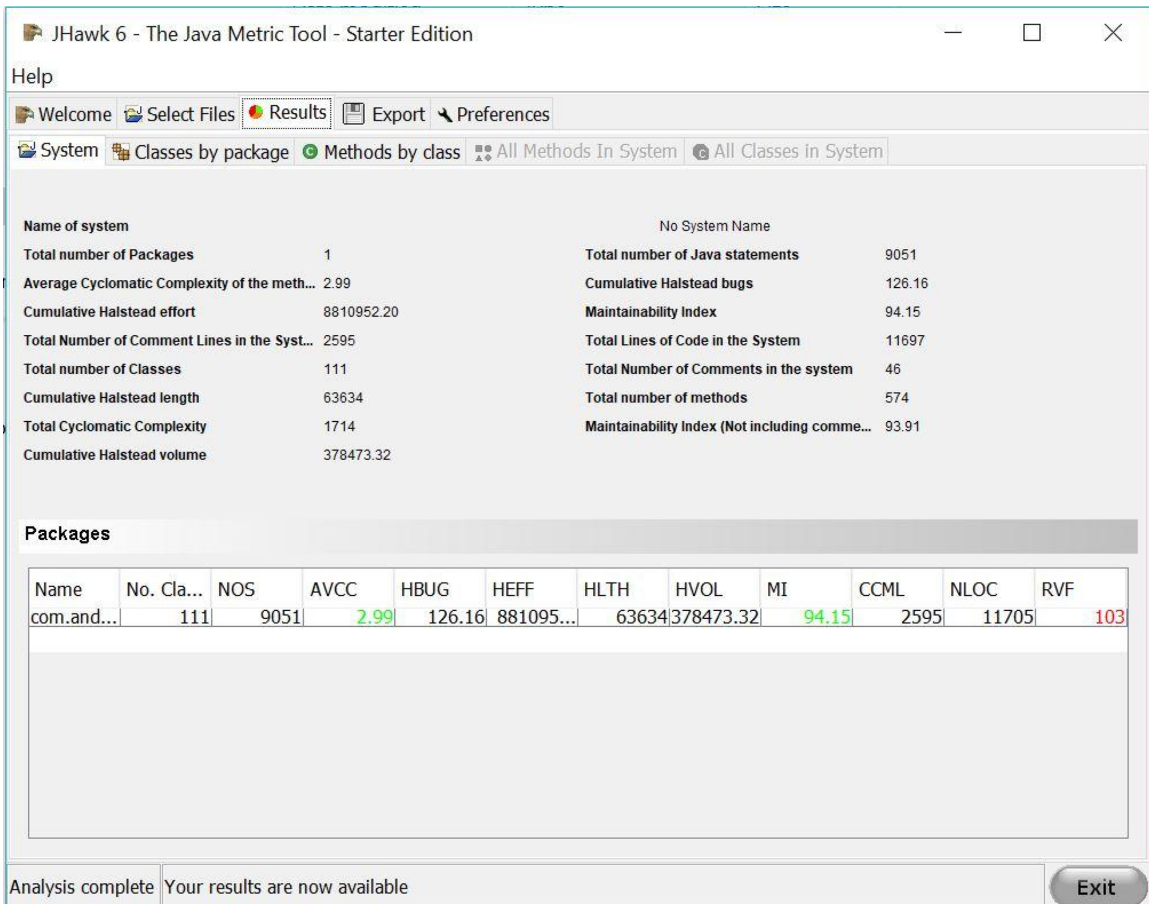


Name	No. Cla...	NOS	AVCC	HBUG	HEFF	HLTH	HVOL	MI	CCML	NLOC	RVF
com.and...	1	17	1.00	0.08	756.54	64	238.85	83.72	21	23	4
com.and...	106	8585	3.07	121.19	886255...	60606	363556.94	92.74	2310	11078	102

3. Froyo Operating System

Froyo version of calendar application has total of 111 classes, 574 methods and 94.15 MI metric. The maintainability index of foryo is degraded. This is due increase in size of system. And the value of refactoring metric is more when compared with Ecliar and gingerbread. This result is deviating from our expectations. And similar results are captured in figure 4. 18 and figure 4. 19. In these figures the refactoring happened in the area where classes had good maintainability. The reason behind it is the new feature support and new functionality addition. This result shows that our refactoring calculation is having exceptions.

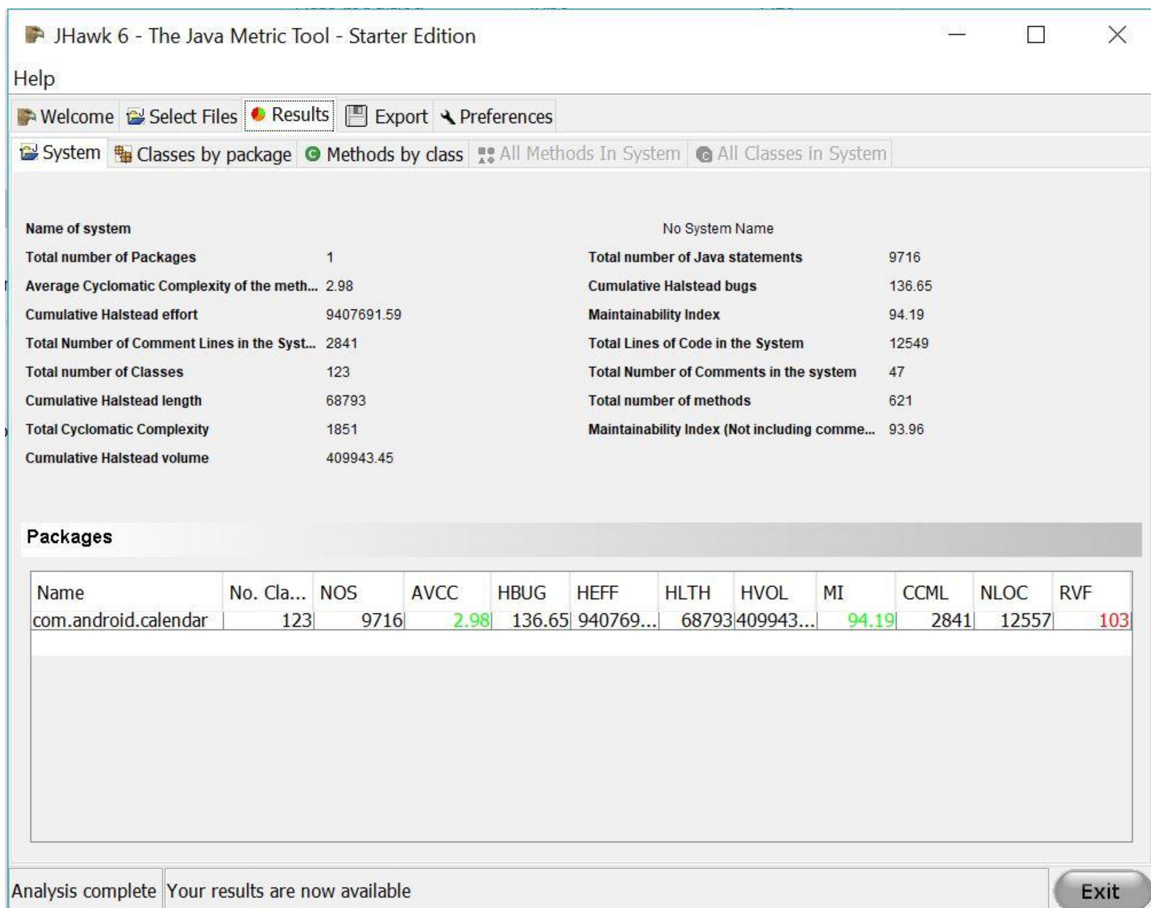
Figure 4. 4 Jhawk output for Froyo



4. Gingerbread Operating System

In gingerbread version of calendar application, we see a increase in SIZE metric of system, number of classes increased to 123, and MI is 94.19,Therefore we can see that SIZE is increased so MI should degrade. But there is no significant change in MI.

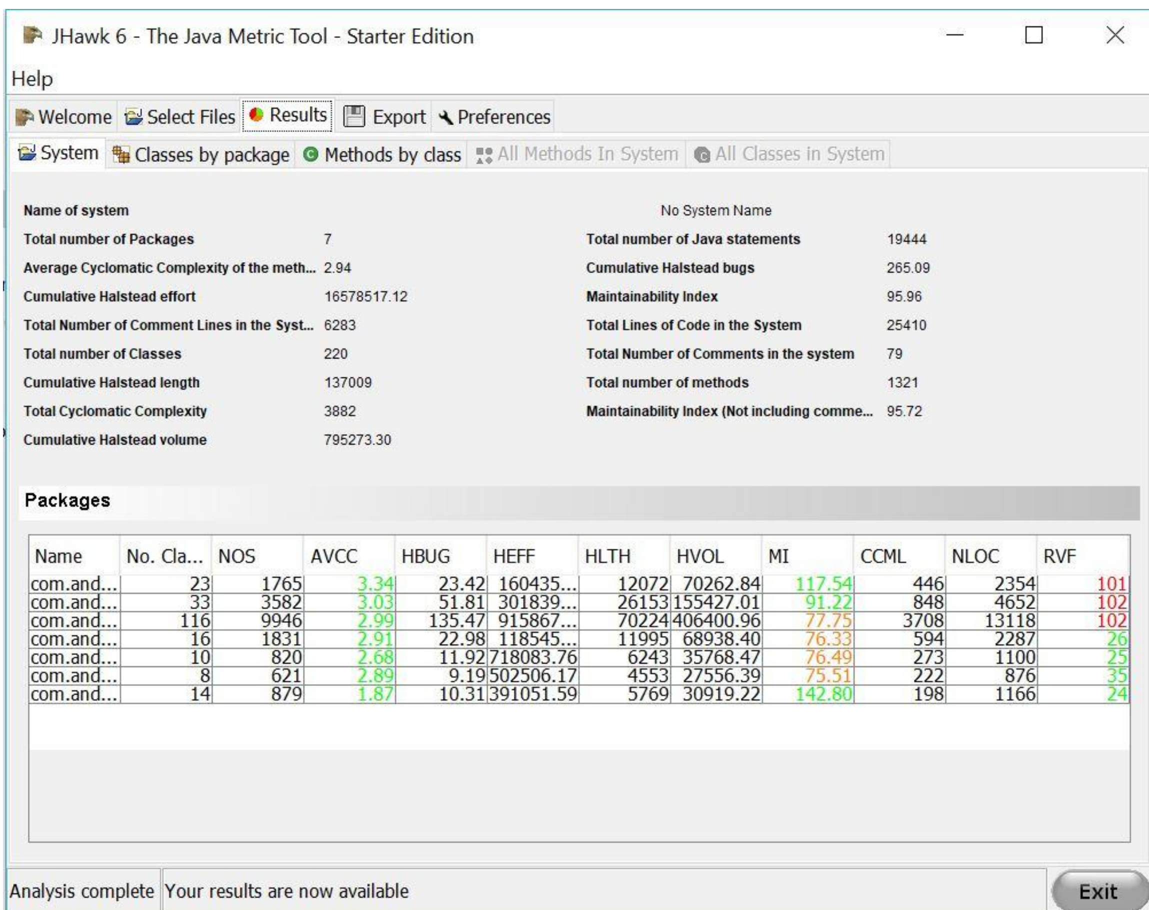
Figure 4. 5 Jhawk output for Gingerbread



5. ICS Operating System

As shown in figure 4. 6, the ICS version of calendar has 220 Classes and 25410 number of lines of code. And MI value is 95.96.Size is increased by significant amount and still there is an improvement in MI. Also the refactoring is also highest in this OS version, due to which MI metric od system uplifted. Therefore we got the expected result.

Figure 4. 6 Jhawk output for ICS

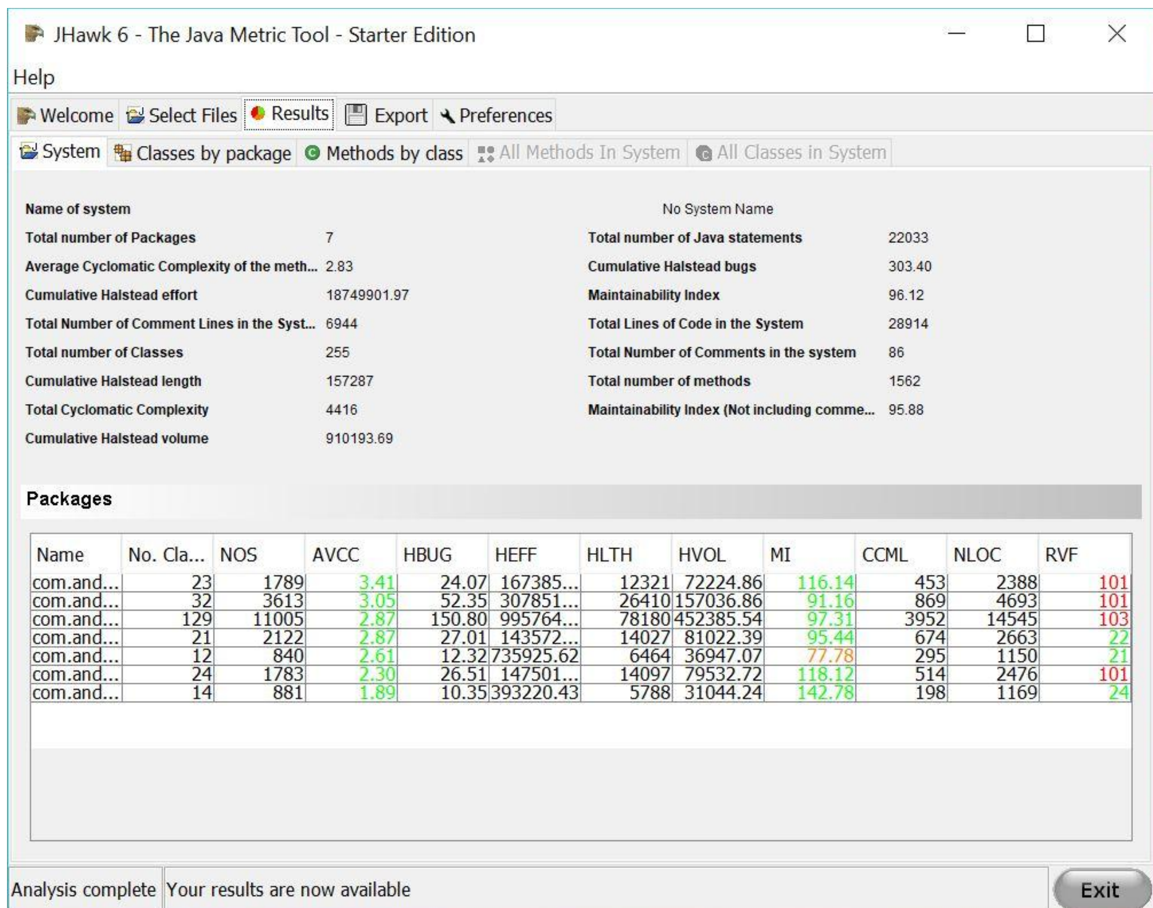


Name	No. Cla...	NOS	AVCC	HBUG	HEFF	HLTH	HVOL	MI	CCML	NLOC	RVF
com.and...	23	1765	3.34	23.42	160435...	12072	70262.84	117.54	446	2354	101
com.and...	33	3582	3.03	51.81	301839...	26153	155427.01	91.22	848	4652	102
com.and...	116	9946	2.99	135.47	915867...	70224	406400.96	77.75	3708	13118	102
com.and...	16	1831	2.91	22.98	118545...	11995	68938.40	76.33	594	2287	26
com.and...	10	820	2.68	11.92	718083.76	6243	35768.47	76.49	273	1100	25
com.and...	8	621	2.89	9.19	502506.17	4553	27556.39	75.51	222	876	35
com.and...	14	879	1.87	10.31	391051.59	5769	30919.22	142.80	198	1166	24

6. JB Operating System

In JB version of Calendar, we see that there is significant changes in SIZE, number of classes became 255 and number of lines of code became 28914.And refactoring is also recorded little less.

Figure 4. 7 Jhawk output for JB



7. Kitkat Operating System

Kitkat version of calendar has MI of 96.02 unit and 324 number of classes. It has 34208 lines of code. But JB had lesser size metric still the MI is almost same, So, we checked the refactoring in this case. From Figure 5.1 2 Kitkat recorded second highest refactoring. Therefore result is also as per our expectation, with the increase in relevant oo metric, the MI metric remained good due to high refactoring value.

Figure 4. 8 Jhawk output for Kitkat

JHawk 6 - The Java Metric Tool - Starter Edition

Help

Welcome | Select Files | Results | Export | Preferences

System | Classes by package | Methods by class | All Methods In System | All Classes in System

Name of system		No System Name	
Total number of Packages	9	Total number of Java statements	25857
Average Cyclomatic Complexity of the meth...	2.86	Cumulative Halstead bugs	360.74
Cumulative Halstead effort	22760307.18	Maintainability Index	96.02
Total Number of Comment Lines in the Syst...	8060	Total Lines of Code in the System	34208
Total number of Classes	324	Total Number of Comments in the system	108
Cumulative Halstead length	185501	Total number of methods	1843
Total Cyclomatic Complexity	5266	Maintainability Index (Not including comme...	95.78
Cumulative Halstead volume	1082229.31		

Packages

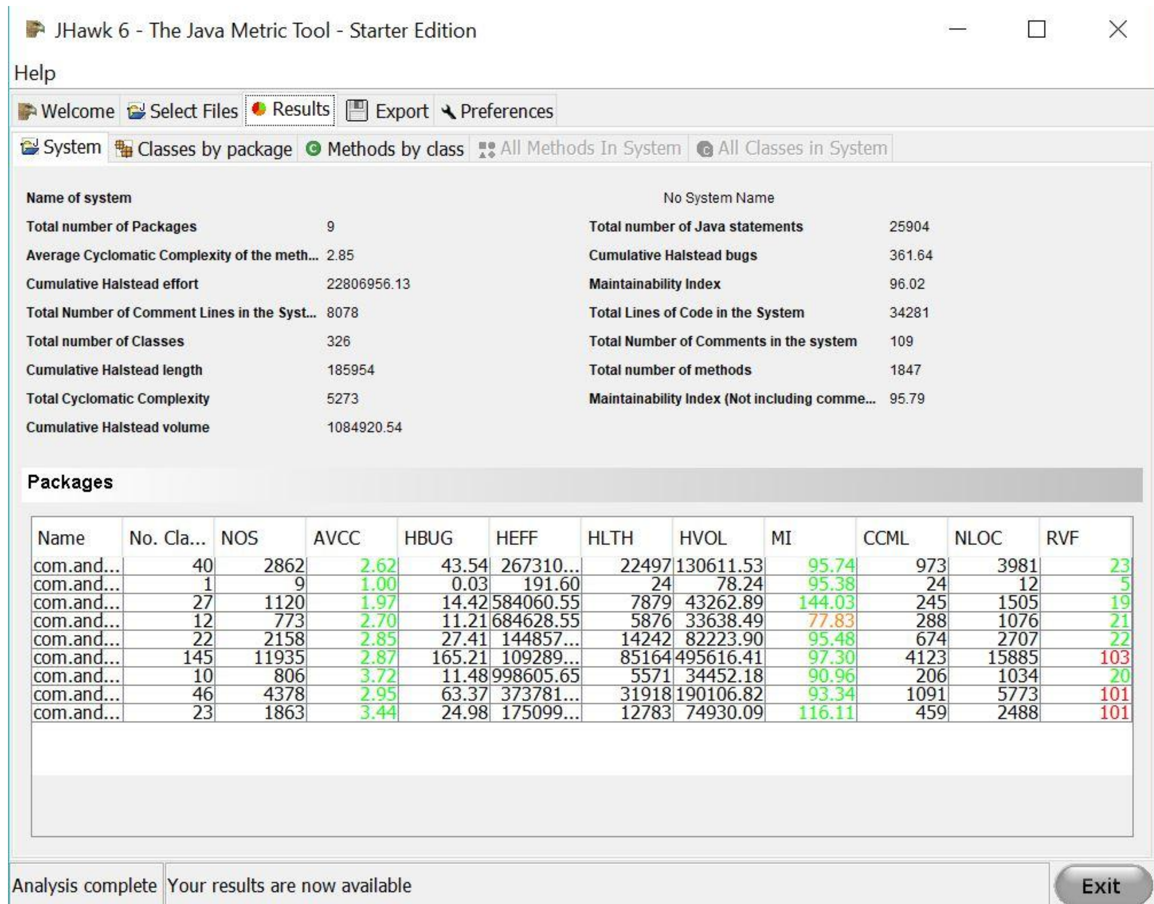
Name	No. Cla...	NOS	AVCC	HBUG	HEFF	HLTH	HVOL	MI	CCML	NLOC	RVF
com.and...	40	2862	2.62	43.54	267310...	22497	130611.53	95.74	973	3981	23
com.and...	1	9	1.00	0.03	191.60	24	78.24	95.38	24	12	5
com.and...	27	1120	1.97	14.42	584060.55	7879	43262.89	144.03	245	1505	19
com.and...	12	773	2.70	11.21	684628.55	5876	33638.49	77.83	288	1076	21
com.and...	22	2158	2.85	27.41	144857...	14242	82223.90	95.48	674	2707	22
com.and...	143	11888	2.87	164.31	108823...	84711	492925.18	97.30	4105	15812	104
com.and...	10	806	3.72	11.48	998605.65	5571	34452.18	90.96	206	1034	20
com.and...	46	4378	2.95	63.37	373781...	31918	190106.82	93.34	1091	5773	101
com.and...	23	1863	3.44	24.98	175099...	12783	74930.09	116.11	459	2488	101

Analysis complete | Your results are now available | Exit

8. Lollipop Operating System

Lollipop has 96.02 MI and 34281 lines of code. The SIZE metric is almost same as kitkat. Also now onwards, the metric data is almost same for all the OS versions, therefore we will limit our study till lollipop version.

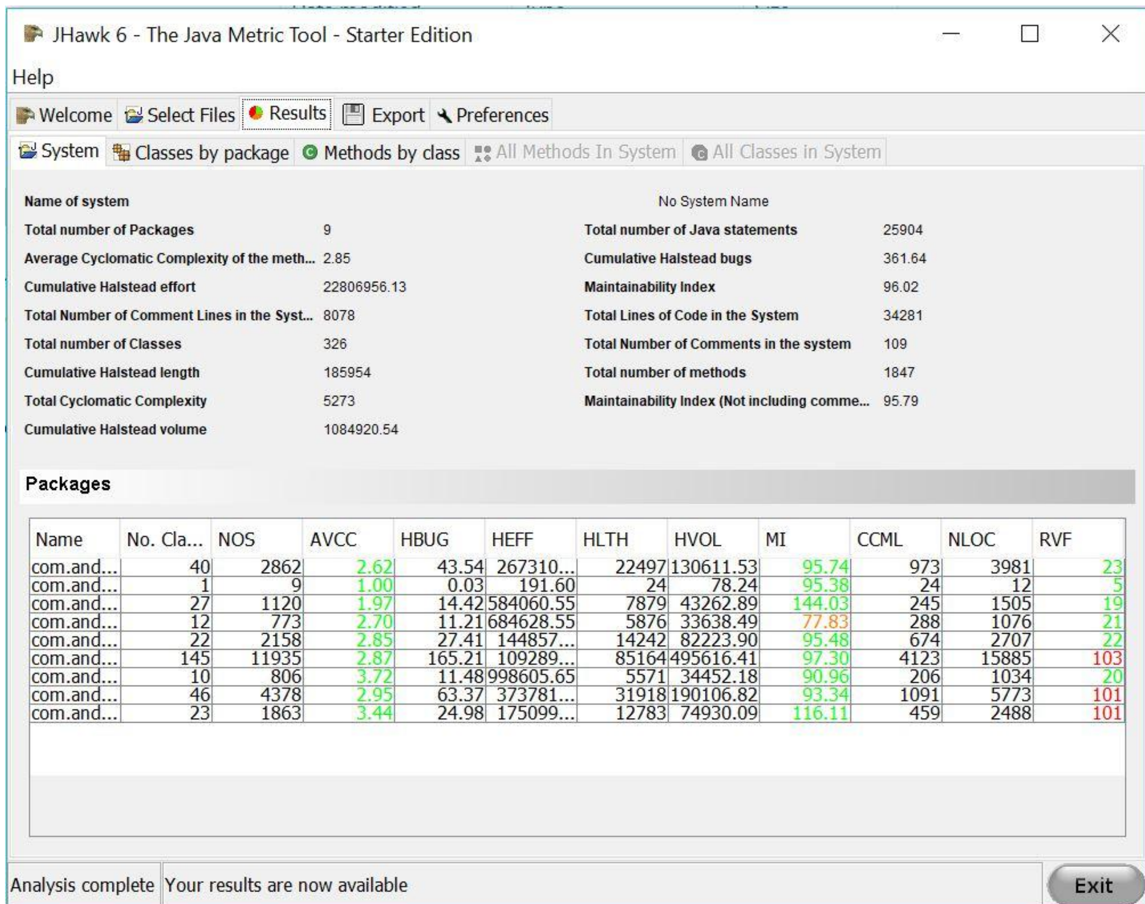
Figure 4. 9 Jhawk output for Lollipop



9. Marshmallow Operating System

The system output w.r.t to OO metric is same as that of previous operating system version .The discussion in Kitkat can be referred.

Figure 4. 10 Jhawk output for Marshmallow

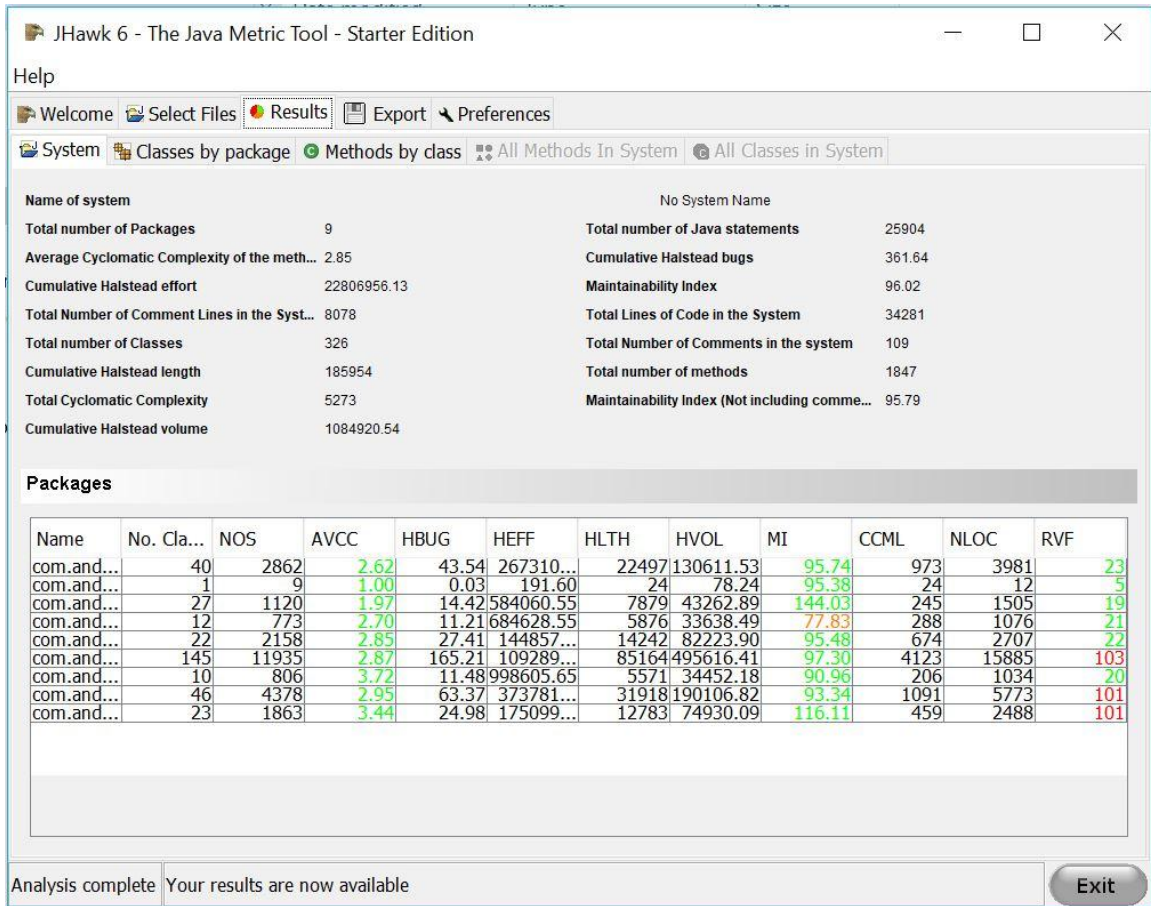


Name	No. Cla...	NOS	AVCC	HBUG	HEFF	HLTH	HVOL	MI	CCML	NLOC	RVF
com.and...	40	2862	2.62	43.54	267310...	22497	130611.53	95.74	973	3981	23
com.and...	1	9	1.00	0.03	191.60	24	78.24	95.38	24	12	5
com.and...	27	1120	1.97	14.42	584060.55	7879	43262.89	144.03	245	1505	19
com.and...	12	773	2.70	11.21	684628.55	5876	33638.49	77.83	288	1076	21
com.and...	22	2158	2.85	27.41	144857...	14242	82223.90	95.48	674	2707	22
com.and...	145	11935	2.87	165.21	109289...	85164	495616.41	97.30	4123	15885	103
com.and...	10	806	3.72	11.48	998605.65	5571	34452.18	90.96	206	1034	20
com.and...	46	4378	2.95	63.37	373781...	31918	190106.82	93.34	1091	5773	101
com.and...	23	1863	3.44	24.98	175099...	12783	74930.09	116.11	459	2488	101

10. Nougat Operating System

Nougat version of operating system is having same metric output as previous version.

Figure 4. 11 Jhawk output for Nougat



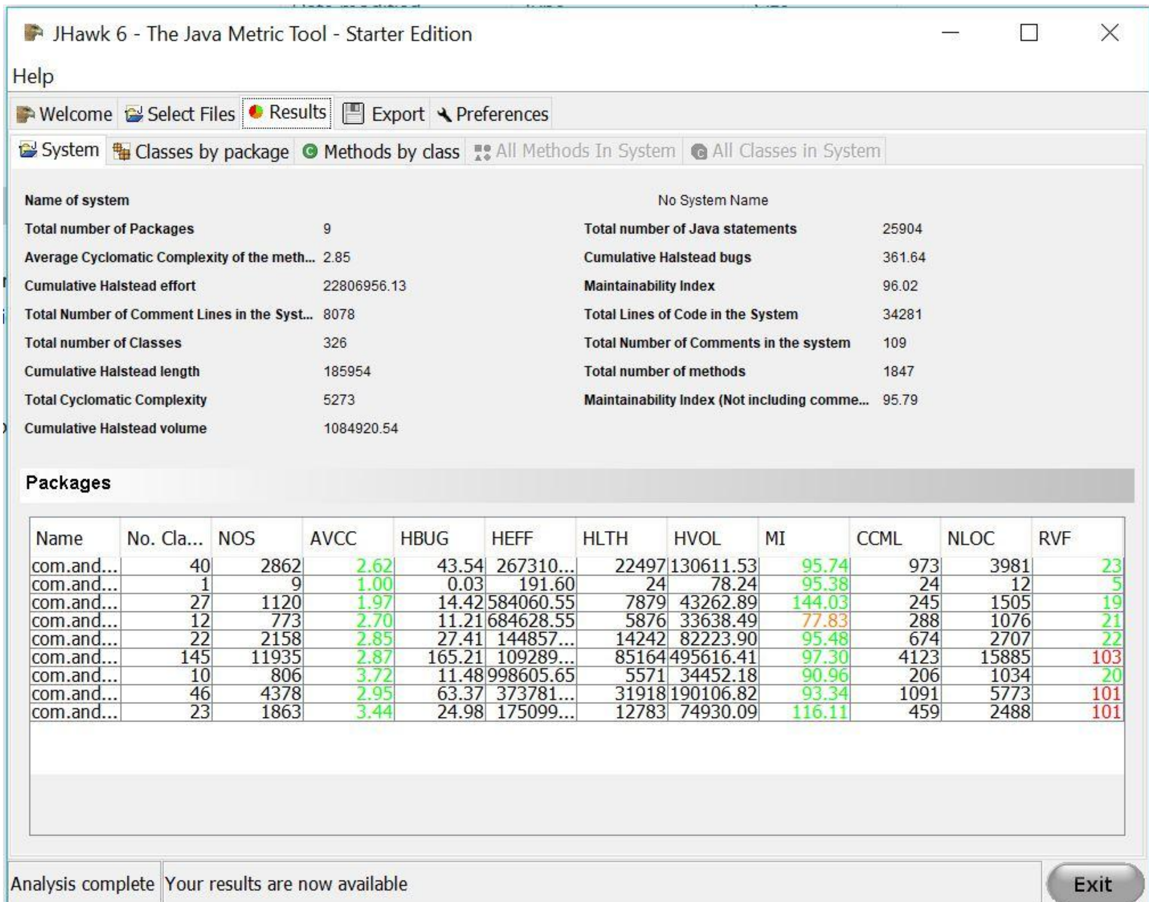
Name of system			No System Name
Total number of Packages	9	Total number of Java statements	25904
Average Cyclomatic Complexity of the meth...	2.85	Cumulative Halstead bugs	361.64
Cumulative Halstead effort	22806956.13	Maintainability Index	96.02
Total Number of Comment Lines in the Syst...	8078	Total Lines of Code in the System	34281
Total number of Classes	326	Total Number of Comments in the system	109
Cumulative Halstead length	185954	Total number of methods	1847
Total Cyclomatic Complexity	5273	Maintainability Index (Not including comme...	95.79
Cumulative Halstead volume	1084920.54		

Name	No. Cla...	NOS	AVCC	HBUG	HEFF	HLTH	HVOL	MI	CCML	NLOC	RVF
com.and...	40	2862	2.62	43.54	267310...	22497	130611.53	95.74	973	3981	23
com.and...	1	9	1.00	0.03	191.60	24	78.24	95.38	24	12	5
com.and...	27	1120	1.97	14.42	584060.55	7879	43262.89	144.03	245	1505	19
com.and...	12	773	2.70	11.21	684628.55	5876	33638.49	77.83	288	1076	21
com.and...	22	2158	2.85	27.41	144857...	14242	82223.90	95.48	674	2707	22
com.and...	145	11935	2.87	165.21	109289...	85164	495616.41	97.30	4123	15885	103
com.and...	10	806	3.72	11.48	998605.65	5571	34452.18	90.96	206	1034	20
com.and...	46	4378	2.95	63.37	373781...	31918	190106.82	93.34	1091	5773	101
com.and...	23	1863	3.44	24.98	175099...	12783	74930.09	116.11	459	2488	101

11. Oreo Operating System

Oreo operating system version of calendar has exactly same system output using Jhawk tool, as Nougat had.

Figure 4. 12 Jhawk output for Oreo



12. Pie Operating System

Pie version of calendar application has same system results as last OS version.

Figure 4. 13 Jhawk output for Pie

JHawk 6 - The Java Metric Tool - Starter Edition

Help

Welcome | Select Files | Results | Export | Preferences

System | Classes by package | Methods by class | All Methods In System | All Classes in System

Name of system		No System Name	
Total number of Packages	9	Total number of Java statements	25904
Average Cyclomatic Complexity of the meth...	2.85	Cumulative Halstead bugs	361.64
Cumulative Halstead effort	22806956.13	Maintainability Index	96.02
Total Number of Comment Lines in the Syst...	8078	Total Lines of Code in the System	34281
Total number of Classes	326	Total Number of Comments in the system	109
Cumulative Halstead length	185954	Total number of methods	1847
Total Cyclomatic Complexity	5273	Maintainability Index (Not including comme...	95.79
Cumulative Halstead volume	1084920.54		

Packages

Name	No. Cla...	NOS	AVCC	HBUG	HEFF	HLTH	HVOL	MI	CCML	NLOC	RVF
com.and...	40	2862	2.62	43.54	267310...	22497	130611.53	95.74	973	3981	23
com.and...	1	9	1.00	0.03	191.60	24	78.24	95.38	24	12	5
com.and...	27	1120	1.97	14.42	584060.55	7879	43262.89	144.03	245	1505	19
com.and...	12	773	2.70	11.21	684628.55	5876	33638.49	77.83	288	1076	21
com.and...	22	2158	2.85	27.41	144857...	14242	82223.90	95.48	674	2707	22
com.and...	145	11935	2.87	165.21	109289...	85164	495616.41	97.30	4123	15885	103
com.and...	10	806	3.72	11.48	998605.65	5571	34452.18	90.96	206	1034	20
com.and...	46	4378	2.95	63.37	373781...	31918	190106.82	93.34	1091	5773	101
com.and...	23	1863	3.44	24.98	175099...	12783	74930.09	116.11	459	2488	101

Analysis complete | Your results are now available | Exit

4.3 Object Oriented metric capturing process

Jhawk tool , is static tool which is used in this study as object oriented metric capturing software. This tools is used as it already been used in multiple study areas[23].The indepth explanation is already done in section 3.1 .The data set generated is verified and and we made some change after manually checking with JArchitect tool outputs.

4.4 Refactoring Dataset construction

The informational collection is done utilizing Jhawk , Ref-finder and Jarchitect Tools. As we have utilized various apparatuses in this study, greatest time utilized in mapping the class information from Ref-discoverer and JArchitect to information caught by Jhawk.The code-repository[22] is downloaded, which is open-source and accessible to everybody. We picked calendar application as source-code to be investigated and analysed. This code is utilized on the grounds that the Calendar application isn't yet researched by any scientist and we needed to check the relationship is same as portrayed in past studies [10,11,13]. We analyzed the outcomes and discovered that outcomes are inclined with past investigations. Jarchitect device is utilized to discover the code-smell and naming standard infringement in code.

Code Smells captured by JArchitect includes types too large, types with so many methods, types with multiple fields, methods too large, so much complex methods with multiple parameters, methods with too many local variables, methods with too many overloads, methods with less comments, types with bad cohesion. These code smells are mapped to the classes and generated the metric. If we have 3 methods as too big code smell which belongs to same class , then we add 3 in too big method code-smell of that class. Now once we have the all the measurements corresponding to code-smells , then we will check for the change in code-smell in next OS versions. If there is any change , we have treated it as related refactoring. This is very critical and time taking activity. Similarly code smells for all the OS versions under study were captured and mapped to the correct class. Then 2 consecutive OS versions are compared and Code smell change is calculated. And this change is treated as one refactoring as per JArchitect tool.

As shown in Figure 4.14 Naming conventions captured in JArchitect are Instance fields should begin with a lower , Interface name should begin with an Upper character , Exception class name should be suffixed with 'Exception' , Types name should begin with an Upper character , Methods name should begin with an lower character , Avoid types with name too long , Avoid methods with name too long , Avoid fields with big name , Avoid prefixing type name with parent package name , Avoid naming types and packages with the same name . These naming convention violation data is also mapped to the classes and then compared with neighbouring version. If name change is observed or classes with similar functionality is added in new version and removed from previous version, then Naming related refactoring is updated for that class and method. This complete process is manual and very time consuming.

4.4.1 Code-Refactoring

When a software's source code is effectively understandable, the software is more maintainable, prompting decreased expenses and enabling valuable advancement assets to be utilized somewhere else. In the meantime, if the code is all around organized, new prerequisites can be presented more productively and with less issues. These two improvement errands, maintainability and upgrade, frequently struggle since new highlights, particularly those that don't fit neatly inside the first outline, result in an expanded support exertion. The re-factoring procedure expects to lessen this contention, by helping non damaging changes to the structure of the source code, keeping in mind the end goal to improve code clearness and viability. Below are some of the refactoring-types which are considered in analyzing the data-set.

- ***Rename***

A method, variable, class or other java thing has a name that is deceiving or befuddling. This requires all references, and conceivably record areas to be refreshed. The way toward renaming a method may incorporate renaming the method in subclasses and customers. Then again, renaming a bundle will likewise include moving documents and catalogs, and refreshing the source control framework.

- ***Move Class***

A Class is in the wrong package, it ought to accordingly be moved to another package where it fits better. All import proclamations or completely qualified names alluding to the given class should be refreshed. The record will likewise must be moved and refreshed in the source control framework.

- ***Extract Method (Long Methods)***

A long method should be separated to upgrade decipher-ability and viability. A segment of code with a solitary legitimate errand is supplanted with a conjuring to another method. This new method is given appropriate parameters, return compose and special cases. By giving the method a demonstrate and enlightening innocence, the first method ends up less difficult to comprehend as it will read like pseudo-code. Extricating the method additionally enables the method to be reused in different spots, impractical when it was tangled among the bigger method. On the off chance that the removed area is well picked, this method might be a characteristic place to change the conduct of the class through subclass, as opposed to a reorder of the current method before rolling out improvements.

- ***Extract Classes and SuperClasses***

A current class gives usefulness that should be altered somehow. A unique class is presented as the parent of the present class, and after that regular conduct is "pulled up" into this new parent. Customers of the current class are changed to reference the new parent class, permitting elective usage (polymorphism). Any methods which are basic to the solid classes are "pulled up" with definitions, while those that will change in subclasses are left unique. And in addition supporting in productive code re-utilize, it

likewise enables new subclasses to be made and utilized without changing the customer classes.

- ***Replace Conditions with Polymorphism (too many conditional statements)***

Methods in a class as of now check some esteem (if or switch statement) keeping in mind the end goal to choose the correct activity to perform. One unimportant illustration is a class that draws a shape, which is characterized by a width and sort (circle or square). The code rapidly winds up confounding as the same if or switch statements are rehashed all through the class, i.e. in methods that compute the territory or edge of the shape. By utilizing polymorphism, the shape particular conduct can be offloaded to subclasses, rearranging the code. This has the additional advantage of permitting different subclasses, e.g. rectangle or star, to be presented without broad code changes.

With every issue over a pretty much evident arrangement has been expressed, as well. Nonetheless, it is obvious to each accomplished programming designer that there are more convoluted code issues, for which straightforward arrangements can not all that effectively be introduced. Clearly, a product designer will normally apply re-factorings effectively just, on the off chance that he/she knows how the product should look like at last. As it were, before attempting to refactor some code, one needs to acquaint oneself with the basic protest arranged outline designs and re-factorings.

Other than above mentioned refactoring, Fields Removed, Methods removed, Classes Removed, Methods Direct Calling, Method indirect Calling and Classes with poor cohesion are also considered in this study. All these refactoring types have self-explanatory names.

4.4.2 JArchitect Tool

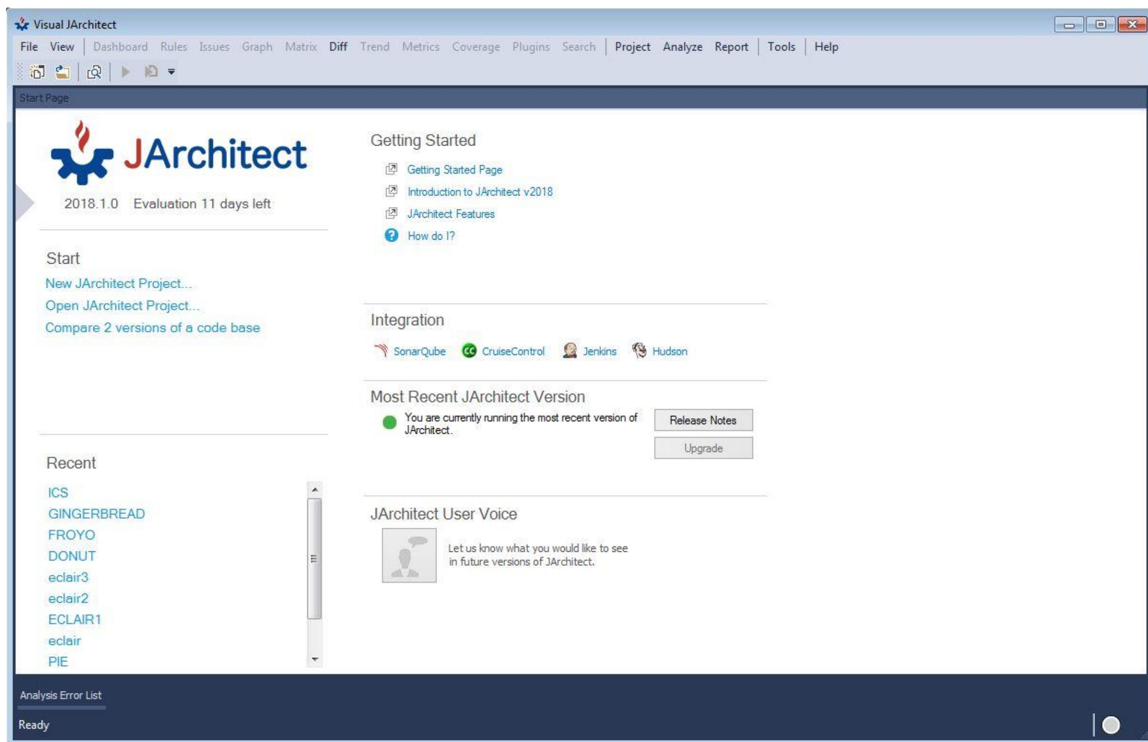


Figure 4. 14 JArchitect Tool

JArchitect is able to tell the developer that over the past hour, the code just written has introduced debt that would cost for example about 30 minutes should it have to be repaid later. Knowing this, the developer can fix the code before even committing it to the source control. With JArchitect code rules are LINQ queries that can be created and customized in a matter of seconds. These queries contain formulas to compute accurate technical debt estimations.The default rule-set offers over a hundred code rules that detect a wide range of code smells including entangled code, dead-code, API breaking changes and bad OOP usage.

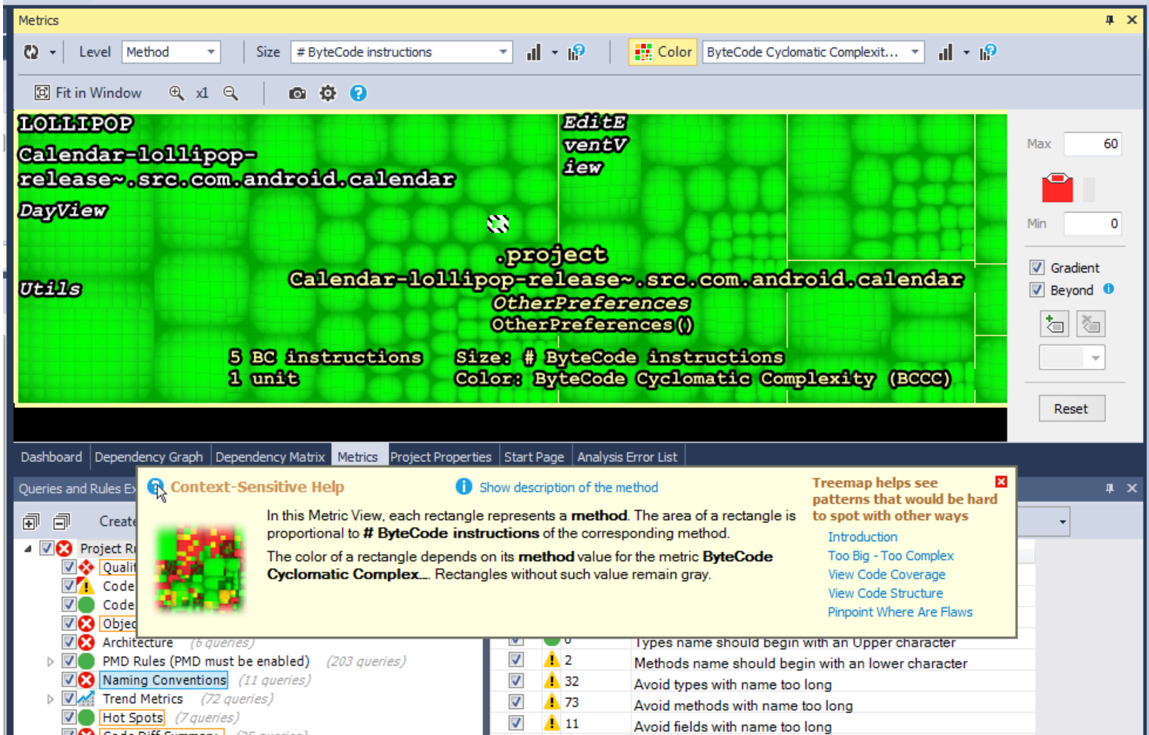


Figure 4. 15 Metric-View of Calendar-Lollipop Version

The Metric view of JArchitect tool shows a method and its size and cyclomatic complexity. So the bigger rectangle in metric view seems to be a code smell, which can be directly access using this metric view.

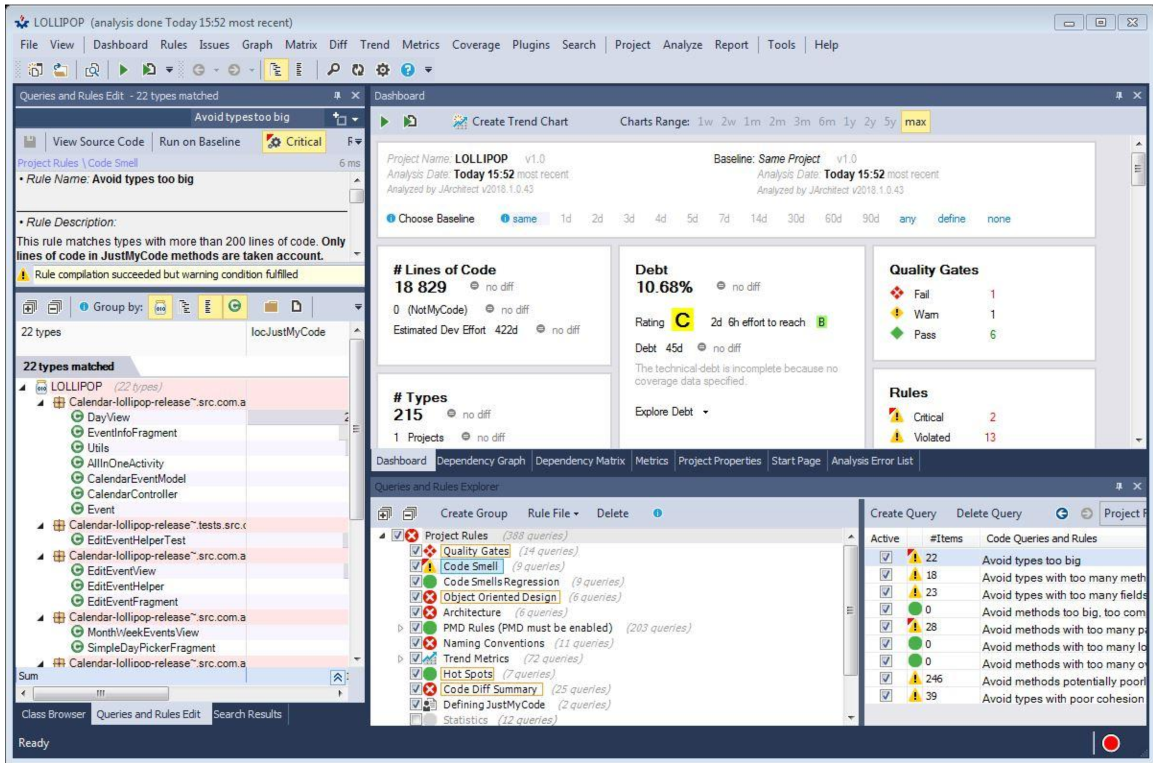


Figure 4. 16 Code-Smells captured by JArchitect of Calendar-Lollipop Version

Code Smells captured by JArchitect included all the events as mentioned in section 4.4, As shown in Figure 4.16.

Name	# Issues	Added	Fixed	Elements	Group
Avoid types too big	16	0	0	types	Project Rules \ Code Smell
Avoid types with too many methods	12	0	0	types	Project Rules \ Code Smell
Avoid types with too many fields	19	0	0	types	Project Rules \ Code Smell
Avoid methods with too many parameters	18	0	0	methods	Project Rules \ Code Smell
Avoid methods potentially poorly commented	197	0	0	methods	Project Rules \ Code Smell
Avoid types with poor cohesion	34	0	0	types	Project Rules \ Code Smell
Nested types should not be visible	3	0	0	types	Project Rules \ Object Oriented Design
Instance fields should begin with a lower character	9	0	0	fields	Project Rules \ Naming Conventions
Methods name should begin with an lower character	2	0	0	methods	Project Rules \ Naming Conventions
Avoid types with name too long	20	0	0	types	Project Rules \ Naming Conventions
Avoid methods with name too long	40	0	0	methods	Project Rules \ Naming Conventions
Avoid fields with name too long	9	0	0	fields	Project Rules \ Naming Conventions
Avoid having different types with same name	11	0	0	types	Project Rules \ Naming Conventions

Table 4. 1 Code-Smell and other Metrics captured

As shown in Table 4.1, Naming conventions captured in JArchitect are Instance fields should begin with a lower , Interface name should begin with an Upper character , Exception class name should be suffixed with 'Exception' , Types name should begin with an Upper character , Methods name should begin with an lower character , Avoid types with name too long , Avoid methods with name too long , Avoid fields with name too long , Avoid having different types with same name , Avoid prefixing type name with parent package name , Avoid naming types and packages with the same identifier .

4.5 Results

In this section, we will discuss about correlation pattern between MI and refactoring. MI model is based on generated data set of OO metrics using Jhawk tool.. The number of classes in donut is 97, eclair has 107, froyo has 111, gingerbread has 123, ICS has 123, JB has 255, kitkat as 342 and all other OS version has 326 classes from lollipop to pie. KitKat onwards, the OO metric data is also almost same, which is evident from Table 5.1. As the number of classes are almost same, in operating system KitKat and above versions, we will consider only Donut, Eclair, Froyo, Gingerbread, ICS, JB and KitKat. The data set is generated using Jhawk , Ref-finder and Jarchitect tools. As we have used multiple tools therefore, maximum time is invested in mapping the class data from Ref-finder and JArchitect to data captured via Jhawk.The code-repository[22] is downloaded, which is open-source and available to everyone. We chose calendar app as source-code to be analysed. This repository is used as, Calendar app software code is not yet analysed by any researcher and we wanted to verify the correlation is same as described in past study [10,11,13]. We compared the results and found that results are inclined with the previous studies. JArchitect tool is used to find out the code-smells and naming rule violations in code. Figure 4. 16 shows the captured information of one of the OS version. After this, we compared the final data-set and the results are discussed in following sections.

4.5.1 DONUT-ECLAIR Correlation

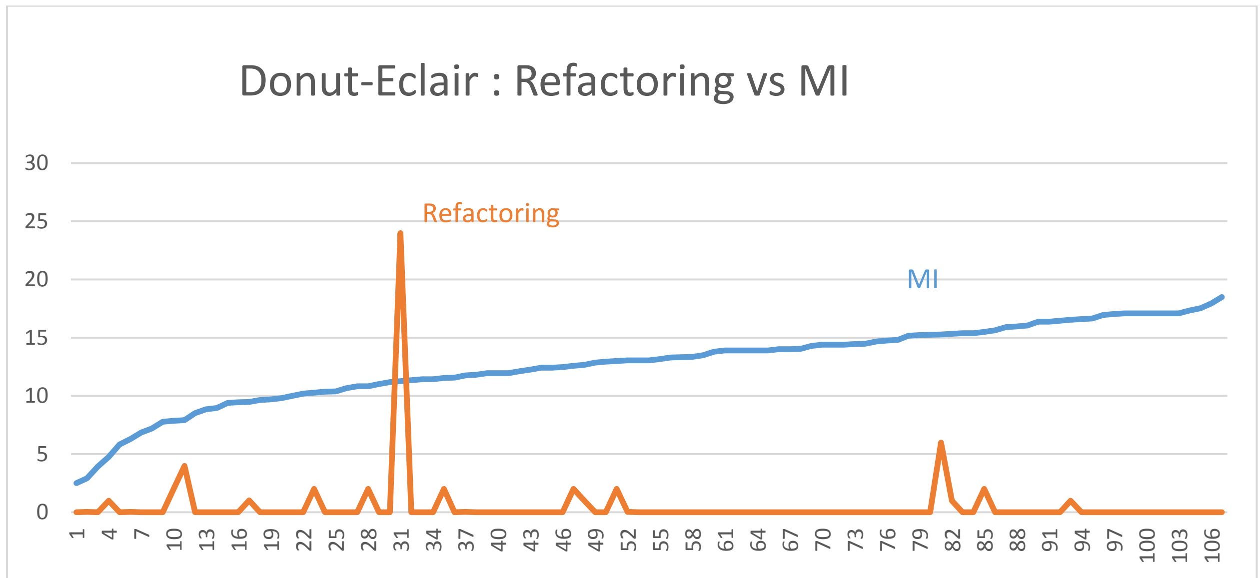


Figure 4. 17 MI and Refactoring of Donut & Éclair

Figure 4.17 shows comparison between MI metric of donut and refactoring of Éclair version of Calendar. We have not considered the negative change in data-set. If negative change is observed, then it is changed to zero in data-base. In almost all the negative cases, either the classes were not present in previous version of calendar or there was a feature enhancement due to which code-smell measurement were increased. Keeping these point in mind, we have mapped the total refactoring with MI metric and found that the classes which were having very poor maintainability are refactored more in its next OS version. Therefore the past studies are verified with positive results. The result of this correlation is shown in Figure 4.17

4.5.2 ÉCLAIR-FROYO Correlation

Figure 4.18 shows comparison between class level MI metric and refactoring. MI in Éclair and refactoring in foryo, shown the same result but 4 to 5 exceptional peaks were observed. When the classes checked in code then it was found that new features were added in Froyo. Due to new functionality the code-smells increased and due to

which refactoring measurement also increased. Therefore our method of finding refactoring has a threat and is not completely valid. Below is the result of correlation with an exception at good maintainability also refactoring happened.

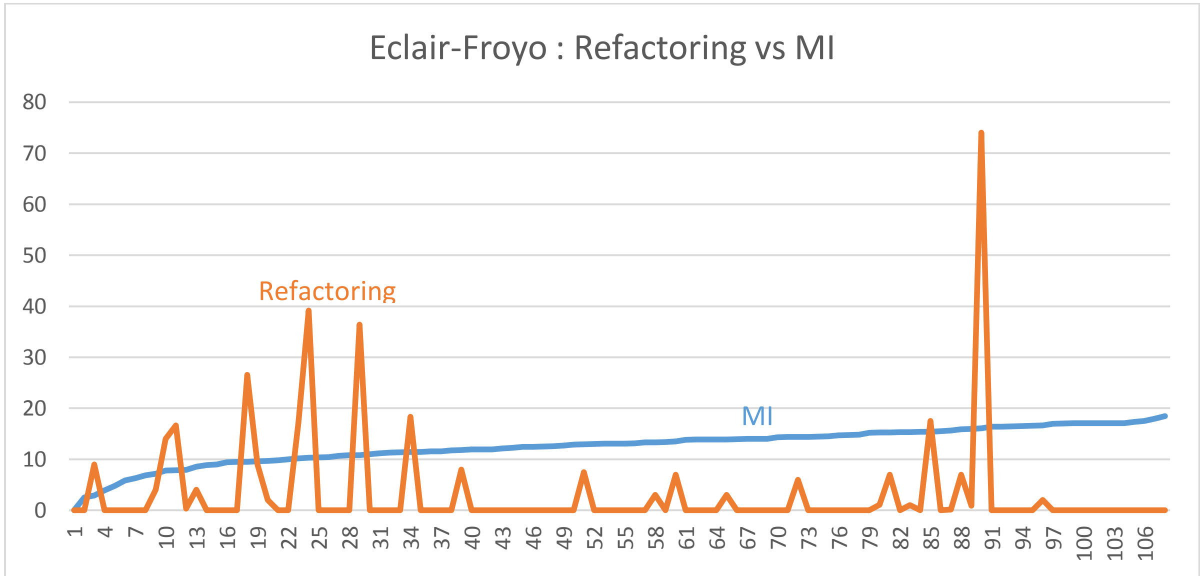


Figure 4. 18 Eclair-Froyo : Refactoring vs MI

4.5.3 FROYO-GINGERBREAD OS Correlation

Figure 4.19 shows comparison between Froyo and Gingerbread, MI metric of froyo is correlated with Gingerbread's refactoring. Similar result is observed with one exception that is the class which is having comparatively good Maintainability is refactored most.

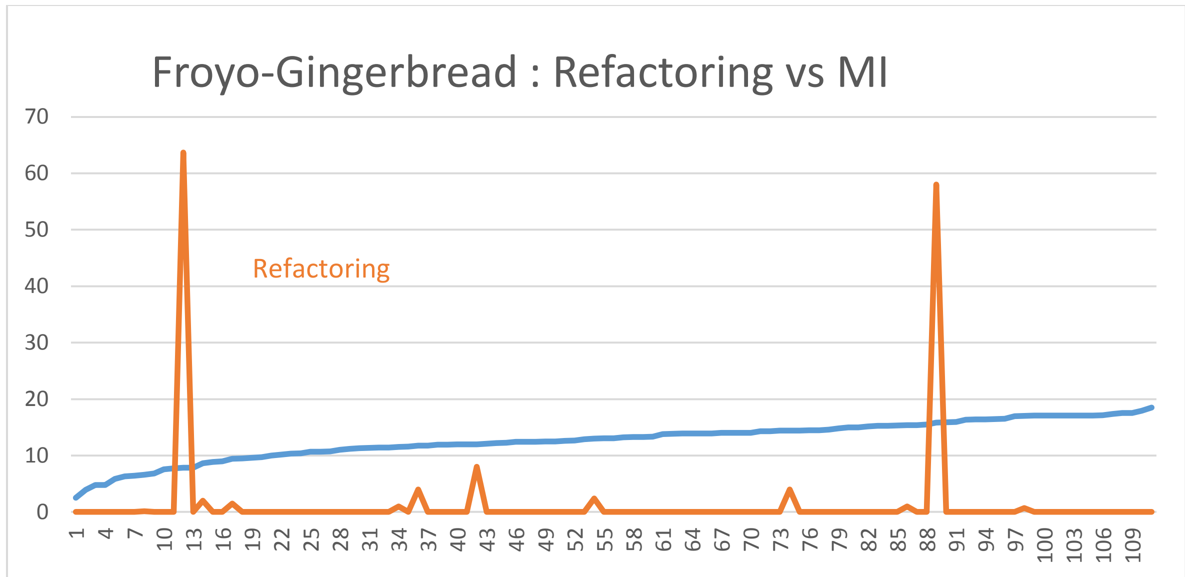


Figure 4. 19 Froyo-Gingerbread : Refactoring vs MI

4.5.4 GINGERBREAD-ICS OS Correlation

Figure 4.20 shows comparison between gingerbread and ICS, MI metric of gingerbread and refactoring of ICS. The results are same, but in this correlation, we have observed that refactoring magnitude is too big, this means ample amount of refactoring were carried out in ICS OS.

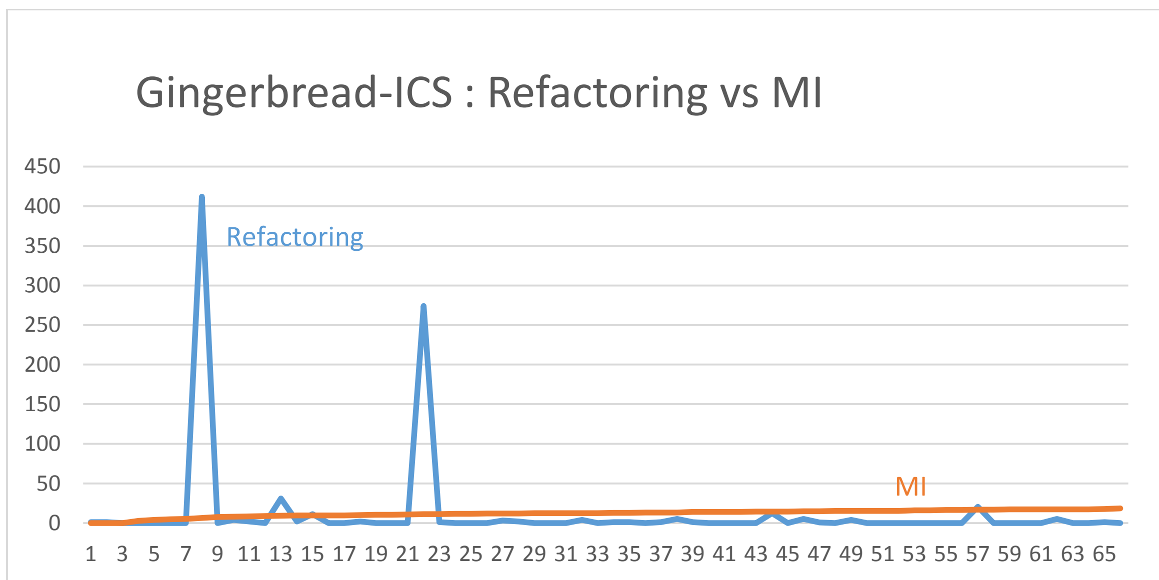


Figure 4. 20 Gingerbread-ICS : Refactoring vs MI

4.5.5 ICS-JB OS Correlation

Figure 4.21 shows comparison between ICS and JB, MI metric of ICS and refactoring in JB. The results are still inline with the previous results. And it seems like Calendar app is showing the expected behaviour as per out study so far, Except for the exceptional peaks seen in the area where maintainability is good.

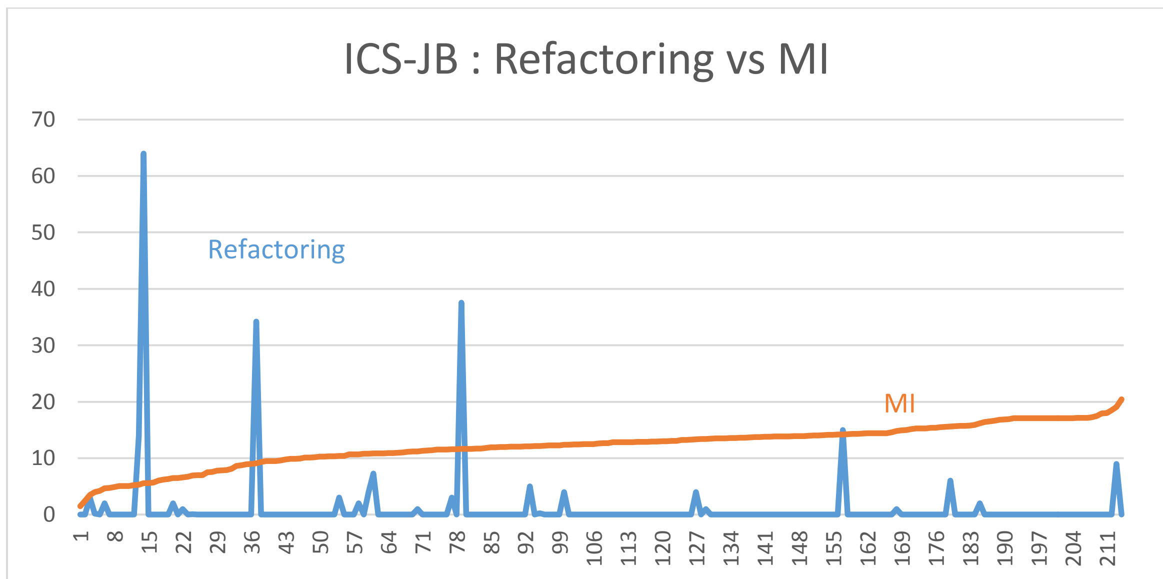


Figure 4. 21 ICS-JB : Refactoring vs MI

4.5.6 JB-KITKAT OS Correlation

Figure 4.22 shows comparison JB and Kitkat OS version of calendar app. The results similar, and those classes are most refactored which were having less maintainability as per MI metric.

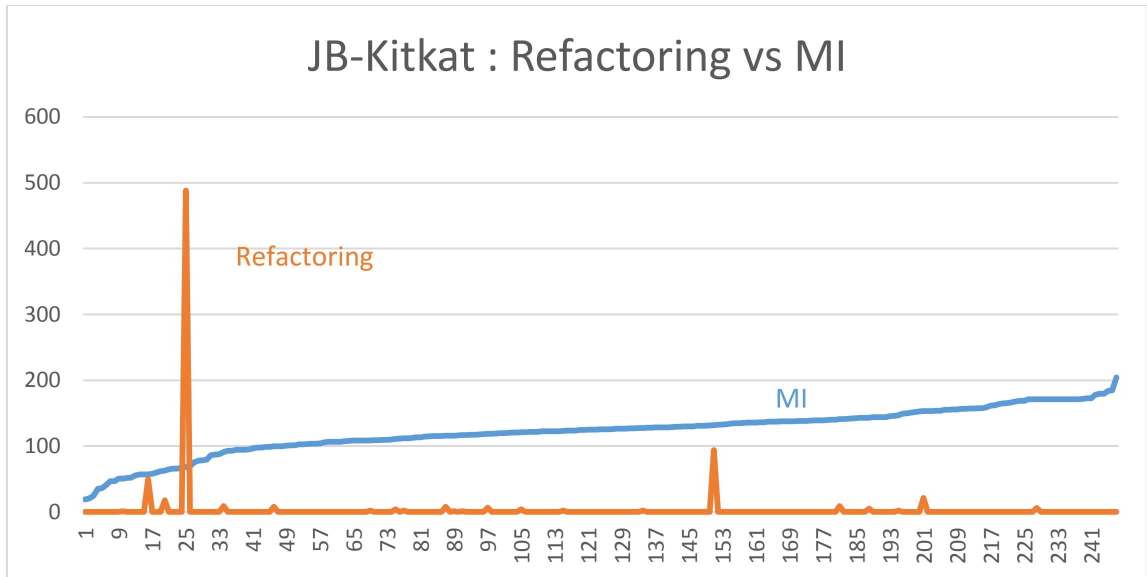


Figure 4. 22 JB-Kitkat : Refactoring vs MI

Chapter 5: Conclusion & Future Work

In Our work we have found relationship among OO Metrics, Code-smells, Refactoring and Maintainability of any class. Finally, the purpose of the examination in this thesis is to add the collection of information about correlation between MI and refactoring. The study of operating system versions of android application Calendar shows that object oriented metrics [2] maintainability [3,4,5] and Code refactoring[14] are closely related. Also multiple OO metrics gets effected by code refactoring including code-smells, naming rules and dead code.

Table 5. 1 System Data

OS Version	Total Number of Classes	Total Cyclomatic Complexity	Total number of Java statements	Cumulative Halstead Bugs	Maintainability Index	Total Line of Code in the System	Total number of Comments in the system	Total number of Methods
Donut	97	1324	7445	104.99	92.91	9588	38	461
Éclair	107	1599	8602	121.27	92.76	11093	42	523
Froyo	111	1714	9051	126.16	94.15	11697	46	574
Gingerbread	123	1851	9716	136.65	94.19	12549	47	621
ICS	220	3882	19444	265.09	95.96	25410	79	1321
JB	255	4416	22033	303.4	96.12	28914	86	1562
Kitkat	324	5266	25857	360.74	96.02	34208	108	1843
Lollipop	326	5273	25904	361.64	96.02	34281	109	1847
Marshmallow	326	5273	25904	361.64	96.02	34281	109	1847
Naugat	326	5273	25904	361.64	96.02	34281	109	1847
Oreo	326	5273	25904	361.64	96.02	34281	109	1847
Pie	326	5273	25904	361.64	96.02	34281	109	1847

The motivation behind the MI was to give a marker of practicality where high MI reflected good maintainability and low MI bad maintainability. Maintainability was examined utilizing OO metric. Eclipse project of Twelve OS versions of the Calendar application, were used as a basis of the this study. The basic inspiration behind why the

relationships were huge seemed, to be a direct result of refactoring of the code segment having very poor MI metric. The below table and graph 5.1 shows the results, that maximum refactoring happened in those classes which has the significantly low maintainability.

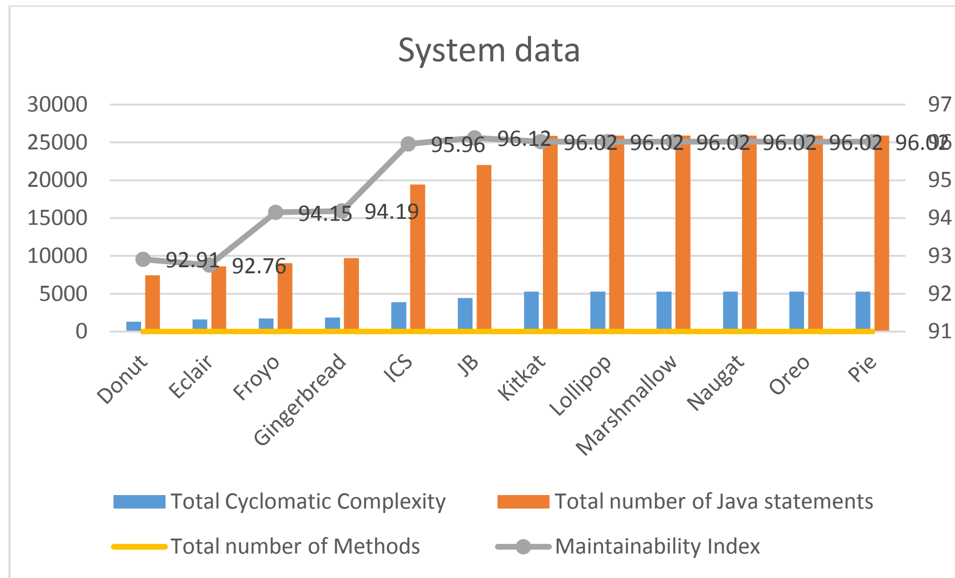


Figure 5. 1 : MI ,TCC,NOM,LOC(java statement) metric plot of 12 OS versions of Calendar app.

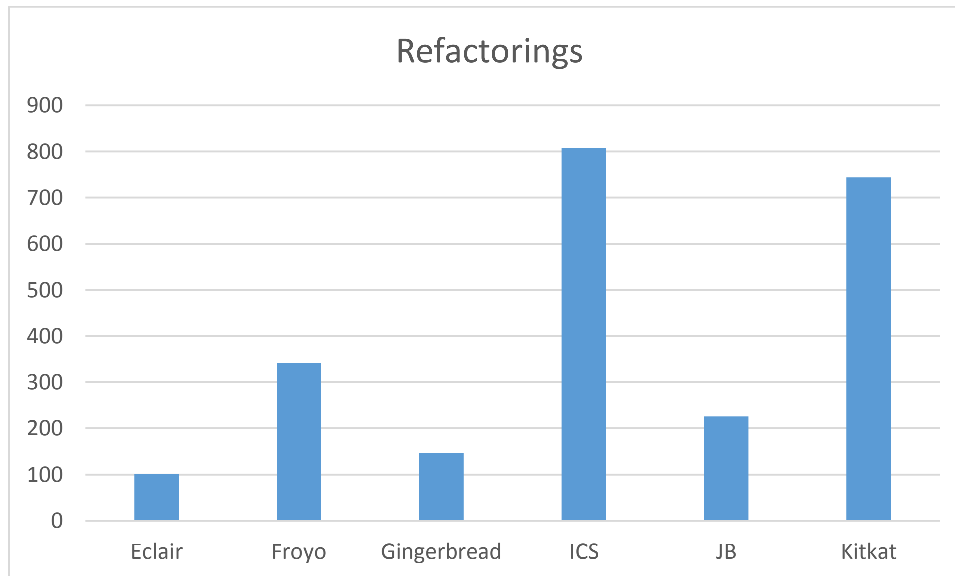


Figure 5. 2 : Total Refactoring in 6 OS versions of Calendar app.

From our experiment, we found that for ICS version, the refactoring is more as compared to other operating system versions and its MI metric is comparatively good when compared with higher end OS version, it is visible in Figure 5.1 and 5.2.

MI and refactoring correlation is significant Hence, we can conclude our work on Refactoring is more in the classes in subsequent releases of Android OS Data sets (like Android Donut to Pie Release),with bad MI metric value.

Bibliography

[1] W. Li, “Another Metric Suite for Object-oriented Programming”,The Journal of System and Software, vol 44, no : 2, pp 155–162,December 1998.

[2] IEEE Standard. 1219-1993 IEEE Standard for Software Maintenance. INSPEC Accession Number: 4493167 DOI:10.1109/IEEESTD.1993.11557 Journal .1993

[3] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," Journal of Systems and Software, vol. 23, no 2,pp. 111-122, 1993

[4] S. R. Chidamber and C. F. Kammerer, "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering, pp. 20, 6, 476-493., 1994.

[5] S. Chidamber, R. Shyam and C. Kamerer, “Towards a metrics Suite for Object-Oriented Design Proceedings”, Proceeding of Conference on object – oriented programming systems, languages and applications, OOPSLA'91, pp.197-211, November, 1991

[6] S.Counsell , X.Liu ,S.Eidh,R.Tonelli, M.Marchesi,G.Concas and A.Murgia, “Re-visiting the 'Maintainability Index' Metric from an Object-Oriented perspective ”, Published in**:** 2015 41st Euromicro Conference on Software Engineering and Advanced Applications , DOI**:** 10.1109/SEAA.2015.41

[7] Baqais,A.,Alshayeb,M.,& Baig,z..A. (2014), " Hybrid intelligent Model for Software Maintenance prediction," Proceedings of World Congress on Engineering, pp 358–362.London,U.K. Springer.

[8] Celia Chen, Shi Lin, Michael Shoga, Qing Wang and Barry Boehm, " How Do

Defects Hurt Qualities? An Empirical Study on Characterizing a Software Maintainability Ontology in Open Source Software," in IEEE International Conference on Software Quality, Reliability and Security (QRS), 2018

[9] S. Counsell, X. Liu, S. Swift,J. Buckley,M. English,S. Herold,S. Eldh and A. Ermedahl, "An exploration of the 'introduce explaining variable' refactoring" published in Proceeding XP '15 workshops Scientific Workshop Proceedings of the XP2015 Article No. 9 and Helsinki, Finland — May 25 - 29, 2015 ACM New York, NY, USA ©2015 table of contents ISBN: 978-1-4503-3409-9

[10] István Kádár, Péter Heged˝us, Rudolf Ferenc and Tibor Gyimóthy, "A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability" ,978-1-5090-1855-0/16 $31.00 © 2016 IEEE DOI 10.1109/SANER.2016.42

[11] István Kádár, Péter Heged˝us, Rudolf Ferenc and Tibor Gyimóthy, "A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability",2016 ACM. ISBN 978-1-4503-2138-9 , DOI: 10.1145/2972958.2972962

[12] Francesca Arcelli Fontana and Stefano Spinelli,"Impact of refactoring on quality code evaluation"Published in:in Proceeding WRT '11 Proceedings of the 4th Workshop on Refactoring Tools Pages 37-40 and Waikiki, Honolulu, HI, USA — May 22 - 22, 2011 ACM New York, NY, USA ©2011 ,table of contents ISBN: 978-1-4503-0579-2 doi>10.1145/1984732.1984741

[13] Ruchika Malhotra and Anuradha Chug, "An Empirical Study to Assess the Effects of Refactoring on Software Maintainability" , 2016 Intl. Conference on Advances in

Computing, Communications and Informatics (ICACCI), Sept. 21-24, 2016

[14] Book : "Refactoring-Improving the Design of Existing Code" by Martin Fowler , https://martinfowler.com/books/refactoring.html

[15] M. Fowler,"Refactoring : Improving the design of existing Code." Addison-Wesley profestional,1999.

[16] Mie Mie ThetThwin and Tong-SengQua,"Application of neural networks for software quality prediction using object-oriented metrics" ‚Journal of Systems and Software Volume 76, Issue 2, May 2005, Pages 147-156

[17] John T. Foreman, Jon Gross, Robert Rosenstein, David Fisher, Kimberly Brune ,"C4 Software Technology Reference Guide: A Prototype" , Software Engineering Institute Carnegie Mellon University,1997

[18] Birgit Geppert, Audris Mockus, and Frank Rößler, "Refactoring for Changeability: A way to go?", 11th IEEE International Software Metrics Symposium (METRICS 2005) 1530-1435/05 $20.00 © 2005 IEEE

[19] R. Malhotra and A. Chug, "Software Maintainability Prediction using Machine Learning Algorithms," in An International Journal (SEIJ), Vol. 2, No. 2, SEPTEMBER 2012

[20] *JHawk tool*, [online] Available: virtualmachinery.com/jhawkprod.htm..

[21] *JArchitect tool*, [online] Available: https://www.jarchitect.com/.

[22] Code-repository : https://android.googlesource.com/platform/packages/apps/Calendar/

[23] http://www.virtualmachinery.com/jhawkreferences.htm